Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

Mixnets e EasyCrypt: Dimostrazioni Crittografiche Assistite da Calcolatore

Tesi di Laurea in Sicurezza e Crittografia

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Fabio Pini

Sessione III
Anno Accademico 2011/2012

Indice

Capitolo 1

Introduzione

1.1 Contesto ed Obiettivi

In ambito crittografico, negli anni si è resa necessaria la definizione formale di criteri di sicurezza, limiti computazionali degli avversari e dei cifrari proposti, di pari passo con lo sviluppo di implementazioni e protocolli pratici.

Per gestire la crescente complessità delle dimostrazioni formali applicate a sistemi crittografici sono stati sviluppati diversi *tool* automatizzati per elaborare prove esatte e basate su codice, nonché per verificare la correttezza di nuove proprietà di sicurezza.

Uno degli ultimi software ad oggi elaborati per dimostrazioni con l'ausilio del calcolatore è EasyCrypt, ancora in fase di sviluppo. L'obiettivo alla base di questo lavoro è stato quella di studiare l'applicazione di EasyCrypt e, più in generale, delle strategie di dimostrazione *game-based* su cui esso si basa ad una tipologia di sistemi di sicurezza pratici, le Mix Networks o Mixnets.

Generalmente i tool suddetti per la costruzione automatica di prove sono utilizzati per dimostrare proprietà formali di sistemi teorici, come ad esempio cifrari a chiave pubblica quali ElGamal o le specifiche di funzioni pseudo-casuali; con EasyCrypt si è tentato invece di verificare l'efficacia di vari strumenti messi a disposizione dal software per dimostrare la correttezza di alcune tecniche impiegate nei protocolli delle mixnets.

Inoltre si è cercato di dare una definizione quantomeno teorica della struttura *game-based* delle dimostrazioni riguardanti le proprietà di sicurezza basilari per le mixnets, in modo da poterla utilizzare in seguito con EasyCrypt.

La struttura della tesi è definita nel modo seguente:

- Il Capitolo 1 è dedicato alla presentazione di nozioni teoriche di base che verranno riprese e trattate nei capitoli successivi;
- Nel Capitolo 2 verranno descritte le mix network in maniera estesa, soffermandosi sui requisiti di sicurezza, la classificazione delle diverse proposte di mixnet esistenti e la presentazione dettagliata delle proposte più rilevanti in letteratura;
- Nel Capitolo 3 verrà meglio definita la teoria alla base delle prove *game-based* e dei prover automatizzati, passando poi a definire le proprietà ed il funzionamento del tool EasyCrypt;
- Nel Capitolo 4 verrà descritto il lavoro svolto per definire la struttura *game-based* delle dimostrazioni delle proprietà di sicurezza delle mixnets e per l'utilizzo di EasyCrypt con alcune di esse, mostrando le funzionalità ancora in fase di sviluppo che potranno essere utili allo scopo;
- Nel Capitolo 5 si descriverà lo stato dell'arte del lavoro, evidenziando le criticità incontrate e traendo conclusioni sui limiti di Easycrypt nell'ambito dei sistemi crittografici pratici e delle mixnets in particolare.

1.2 Teoria di Base

Verranno descritte ora alcune definizioni riguardanti tecniche crittografiche che verranno utilizzate nei capitoli successivi.

1.2.1 Sistemi Crittografici a Chiave Pubblica

Definiamo sistema crittografico a chiave pubblica o asimmetrica una tripla (KG, Enc, Dec) di algoritmi PPT (*probabilistic polynomial time*):

- KG è l'algoritmo di generazione delle chiavi pubblica pk e privata sk:

$$(pk, sk) \xleftarrow{\$} KG(1^k)$$

dove k è il parametro di sicurezza;

- *Enc* è l'algoritmo di crittazione del plaintext m con chiave pubblica pk. Può esere randomizzato con un valore $r \in R_{pk}$:

$$c = Enc_{pk}(m; r)$$

- *Dec* è l'algoritmo di decrittazione di un crittogramma c crittato con Enc_{pk}, utilizzando la chiave privata sk in maniera deterministica:

$$m = Dec_{sk}(c)$$

Asseriamo che un sistema crittografico a chiave pubblica sia CPA-sicuro, ossia resistente ad attacchi *chosen plaintext* se rispetta la definizione di Katz-Lindell in [1]. Nel *CPA indistinguishability experiment* ad un avversario PPT $\mathcal{A}$ vengono fatti scegliere due plaintext m_0 ed m_1 e gli viene presentato un crittogramma c che rappresenta la crittazione di uno dei due plaintext a caso. Se l'avversario non riesce a capire con probabilità trascurabile maggiore a $\frac{1}{2}$ (pari ad una scelta casuale) di quale plaintext c è la crittazione, lo schema è detto CPA-sicuro.

Per trascurabile si intende effettivamente *negl* ovvero *negligibile*: per ogni costante c la probabilità di successo dell'avversario è inferiore a n^c per valori sufficientemente grandi del parametro di sicurezza n.

Si noti come un sistema crittografico con crittazione deterministica non può essere CPA-sicuro, in quanto l'avversario può semplicemente crittare ogni input m_0 e m_1 ed individuare subito c. Nelle mixnets vengono impiegati principalmente i sistemi crittografici RSA ed ElGamal.

1.2.2 RSA

RSA (Rivest, Shamir, Adelman) è un algoritmo basato sulla complessità computazionale della fattorizzazione in numeri primi. Nella versione originale, detta *raw RSA*, l'algoritmo non è CPA-sicuro, a causa della crittazione deterministica. Nelle mixnets viene usata una versione con padding random r e *rerandomization*.

KG è un algoritmo che fa uso di operazioni in aritmetica modulare: sceglie a caso due grandi numeri primi p e q, calcola il cosiddetto 'modulo' $n = pq$ e genera e tale che sia più piccolo di n e coprimo con $(p-1)(q-1)$ e d tale che $e * d \equiv 1\ (mod(p-1)(q-1))$: pk sarà (n, e) e sk sarà (n, d).

La crittazione è così definita:

$$Enc(m;r) = (m||r)^e \, mod \, n$$

dove $m||r$ è il plaintext di cui viene effettuato il padding con una stringa random r.

La decrittazione si ottiene calcolando:

$$Dec(c) = ((m||r)^e)^d \, mod \, n = (m||r)^{ed} \, mod \, n = (m||r)$$

1.2.3 ElGamal

ElGamal è un sistema a chiave pubblica definito su un gruppo ciclico e la cui sicurezza è dipendente dall'assunzione DDH [1]. La versione qui descritta è *semantically secure* o CPA-sicura.

Vengono generate le chiavi a partire dal sottogruppo di ordine q del gruppo ciclico $\mathbb{Z}_p^*$, dove p è un numero primo e q è un altro grande numero primo, fattore di $p-1$. Otteniamo così:

$$\begin{cases} sk = x \xleftarrow{\$} G \\ pk = y = g^x \, mod \, p \end{cases}$$

in cui g è un generatore del gruppo G e $G \subset \mathbb{Z}_p^*$ ha ordine $|G| = q$.

La crittazione è data da:

$$c = Enc_{pk}(m;r) = (\alpha, \beta) = (g^r, m * y^r)$$

dove r è un padding random appartenente al gruppo G.

La decrittazione è definita come:

$$m = \frac{\beta}{\alpha^x}$$

1.2.4 Proprietà di ElGamal

ElGamal presenta la proprietà di omomorfismo: date due crittazioni $Enc_{pk}(m_1; r_1)$ e $Enc_{pk}(m_2; r_2)$, è possibile ottenere $Enc_{pk}(m_1 m_2; r_1 + r_2)$ senza decrittare individualmente m_1 ed m_2. ElGamal mostra omomorfismo moltiplicativo, dato che moltiplicando modularmente due crittazioni otteniamo:

$$Enc_{pk}(m_1; r_1) \odot Enc_{pk}(m_2; r_2) =$$

$$(g^{r_1}, m_1 * y^{r_1}) \odot (g^{r_2}, m_2 * y^{r_2}) =$$

$$(g^{r_1+r_2}, (m_1 m_2) * y^{r_1+r_2}) =$$

$$Enc_{pk}(m_1 m_2; r_1 + r_2)$$

Dalla proprietà di omomorfismo deriva la proprietà di reencryption:

$$RE_{pk}(c; r') = (g^r * g^{r'}, m * y^r * y^{r'}) = Enc_{pk}(m; r + r')$$

con la quale a partire da un crittogramma c è possibile generare altri crittogrammi dello stesso plaintext m corrispondente utilizzando ogni volta padding di randomizzazione diversi.

ElGamal possiede una variante *threshold*, proposta da Desmedt e Frankel [2] con la quale è possibile suddividere la chiave segreta e quella pubblica tra un qualsiasi numero k di partecipanti.

All'atto pratico, la chiave segreta x viene suddivisa in parti $x_1, x_2, ...x_l$ secondo la suddivisione Shamir k-out-of-l [3] ed a ogni x_i è associata una parte di chiave pubblica $y_i = g^{x_i}$. Ogni partecipante mantiene segreta agli altri la propria x_i che viene usata al momento di effettuare la decrittazione condivisa (da una soglia di almeno k partecipanti):

$$m = \frac{\beta}{\prod \alpha^{x_i \lambda_i(0)}}$$

dove α^{x_i} è calcolato da ogni partecipante indipendentemente e $\lambda_i(0)$ è il coefficiente di interpolazione polinomiale di Lagrange del partecipante i-esimo.

1.2.5 Prove Zero-Knowledge

Le prove zero-knowledge o ZK, introdotte da Goldwasser [4] sono protocolli interattivi tra due agenti, un verifier ed un prover. Il prover deve convincere il verifier che conosce alcune informazioni riguardanti un certo segreto, ma senza rivelare nulla del segreto stesso. Per fare questo, il verifier invia alcune *challenges* al prover, che risponde. Il verifier controlla le risposte e decide se il prover conosce effettivamente le informazioni fornite riguardanti il segreto.

Possiamo definire perciò un protocollo *challenge-response* come in Fig. 1.1: il prover esprime la propria conoscenza sul segreto mediante uno *statement*, poi il verifier invia messaggi di challenge a cui il prover risponde con messaggi di response, prima che il verifier decida se lo statement sia vero e quindi se il prover sia onesto, o al contrario se lo statement sia falso e quindi se il prover sia disonesto. La complessità sta nel definire il protocollo in modo che nessuna informazione, oltre alla verità sullo statement, venga passata al verifier.

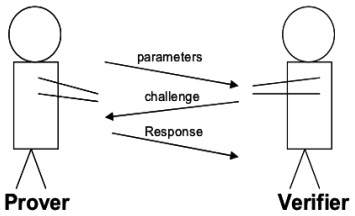

Fig. 1.1: Un protocollo challenge-response

Possiamo affermare che le prove ZK devono soddisfare tre proprietà:

- **Completeness**: il protocollo di prova è completo se e solo se un prover riesce a convincere un verifier della propria onestà nel caso in cui lo statement sia vero;

- **Soundness**: se lo statement è falso, il prover riesce a convincere il verifier della propria onestà con probabilità trascurabile;

- **Zero-Knowledge**: se lo statement è vero, nessun verifier-avversario PPT ottiene informazioni, a parte la veridicità dello statement. In pratica, qualunque cosa produca in output dopo il protocollo il verifier, avrebbe potuto produrla anche prima di interagire con il prover.

Alcune prove ZK, che solitamente sono quelle impiegate dalle mixnets, soddisfano una proprietà aggiuntiva, quella di estrazione, mediante la quale il prover può ricavare il segreto a partire da qualunque statement che lo riguarda.

Le prove che possiedono tale proprietà sono dette prove ZK di conoscenza ed implicano la conoscenza dell'intero segreto e non solo di informazioni che lo riguardano.

1.2.6 Funzioni Hash

Le funzioni *hash* sono in generale funzioni che prendono in input stringhe di dimensione arbitraria ed eventualmente una chiave e producono in output stringhe di dimensione inferiore.

Definiamo funzioni *hash one way* quelle funzioni hash che possono essere computate in tempo polinomiale (*easy to compute*), ma sono *hard to invert*: un eventuale avversario PPT riesce ad invertire tali funzioni con probabilità trascurabile. Per la definizione formale delle funzioni hash e le relative nozioni di sicurezza si rimanda al manuale [1].

Nell'ambito delle mixnets, le funzioni hash vengono utilizzate per la costruzione di MAC (*message authentication code*) o *checksum* per la verifica dell'integrità, nonché per la creazione di chiavi simmetriche nel caso delle mixnets ibride.

Capitolo 2

Mix Networks

2.1 Mix Network: una Vista ad Alto Livello

Il lavoro che segue è riferito a definizioni formali e protocolli descritti a livello teorico; applicazioni sperimentali non verranno prese in considerazione, concentrando l'esposizione sui concetti fondamentali e sulle tecniche ideate.

2.1.1 Definizione Generale e Topologia

Una definizione formale di mixnet 'ideale' è la seguente:

> Una mixnet è un sistema *multistage* che prende in input una serie di elementi e produce in output un nuovo *batch* contenente gli elementi iniziali, trasformati con strumenti crittografici e permutati. La permutazione combinata alla crittazione consente di ottenere la proprietà di *untraceability* tra input ed output.

In Fig. 2.1 vediamo un esempio di stage o *mix server*:

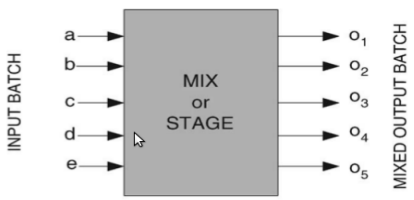

Fig. 2.1: Stage di una mixnet: gli input sono restituiti permutati in output

Le mixnets sono *multistage* in quanto si utilizzano un certo numero di server (corrispondenti agli *stage*) interconnessi, a seconda del grado di anonimità richiesto. Ogni server, come da definizione, effettua una permutazione e qualche operazione crittografica sugli input producendo un diverso batch di output. In Fig. 2.2 possiamo vedere rappresentate due topologie diverse di mixnet:

- a) **Topologia a cascata**: si ha una sequenza fissa di stage interconnessi e tutti gli input passano attraverso ogni stage presente;
- b) **Topologia Free-Routing**: ogni input può passare attraverso uno o più stage tra quelli presenti.

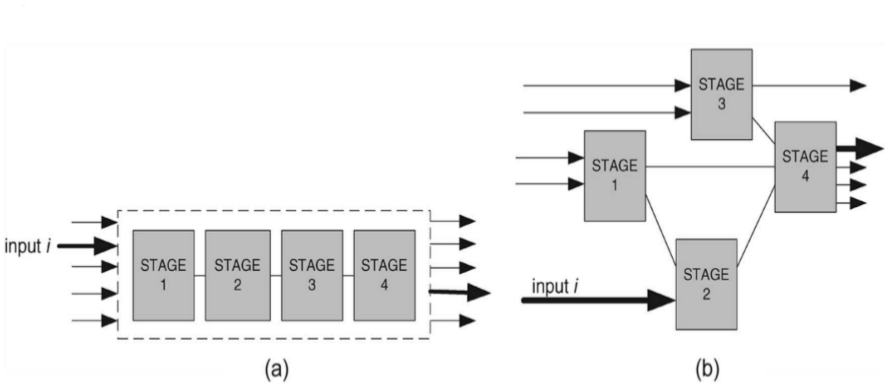

Fig. 2.2: Topologie di mixnet: a) Cascade b) Free-Routing

Le proposte descritte in letteratura sono principalmente con topologia a cascata: quelle che verranno descritte successivamente saranno di questa topologia.

Le mixnets a topologia free-routing sono impiegate principalmente in campi come quello delle telecomunicazioni anonime o via web.

2.1.2 Requisiti di Sicurezza delle Mixnets

Le mixnets devono rispettare una serie di requisiti diversi, a partire ovviamente da quelli di sicurezza:

- **Correttezza**: per dimostrare la correttezza di una mixnet occorre che siano soddisfatte due condizioni:

1. Ad ogni stage vengono crittati (e/o decrittati a seconda dei casi) correttamente gli input;
2. Una funzione di permutazione, in genere *pseudorandom*, mescola casualmente i crittogrammi in output.

Se entrambe vengono rispettate, allora si verifica che ad ogni output corrisponde esattamente un input, facendo sì che la mixnet si comporti come una funzione bigettiva. Inoltre è impossibile che due output siano ottenuti dallo stesso input. Se tutti gli stage sono onesti, tutti gli output sono corretti;

- **Integrità**: gli input ad ogni stage non possono essere aggiunti, cancellati o manipolati;
- **Anonimità**: come già accennato, la proprietà di anonimità è garantita da quella di untraceability, ossia dalla impossibilità per qualunque avversario di associare gli input della mixnet ai suoi output;
- **Verificabilità**: ogni mixnet deve dare prove di correttezza, generalmente di tipo ZK, del mescolamento;
- **Robustezza agli attacchi**: la mixnet deve attivare meccanismi per resistere ad attacchi provenienti dagli avversari, intesi in senso crittografico. Vedremo nella prossima sezione il modello per gli attacchi disponibile per le mixnets;
- **Fault Tolerance**: la mixnet deve poter funzionare correttamente anche in presenza di alcuni stage guasti o corrotti durante lo svolgimento delle operazioni. Solo alcune mixnets rispettano questo requisito.

Abbiamo poi alcuni requisiti implementativi, per rendere operative le mixnets in contesti pratici:

- **Scalabilità**: il numero di input per una mixnet può variare sensibilmente a seconda del contesto in cui è utilizzata; oltre a questo, la mixnet deve sapere gestire un aumento del numero di stage in caso di requisiti di anonimità più stringenti;
- **Efficienza**: occorre minimizzare le computazioni più complesse ed il numero di operazioni crittografiche da eseguire, riducendo lo sforzo computazionale necessario;

- **Performance**: in ambiti come quello delle telecomunicazioni anonime, vi sono requisiti volti ad assicurare *Quality of Service* delle comunicazioni, tenendo conto di parametri come latenza e *throughput*;

2.1.3 Modello dell'Avversario per Mixnets

Gli attacchi di un ipotetico avversario possono essere di due tipi diversi, attivi o passivi.

> **Attacchi passivi**: prevedono che un avversario non interferisca con le operazioni delle mixnets, ma si limiti ad osservare i dati a disposizione nel tentativo di correlare gli input agli output, violando così l'anonimità della mixnet.

Un attacco di questo genere è il *traffic analysis attack*, in cui l'avversario osserva solamente il traffico dei dati in input/output dai vari stage. Aumentando il numero di elementi l in input alla mixnet le probabilità di riuscita dell'avversario diminuiscono.

> **Attacchi attivi**: prevedono l'intervento dell'avversario per corrompere gli input della mixnet e poi la violazione della sua integrità abilitando il *tracing* degli input corrotti, cercando conferme sulla correlazione tra input ed output. Il traffico può essere corrotto aggiungendo, togliendo o modificando input ad alcuni stage.

Vedremo come anche attacchi che inviano copie di un input possano risultare pericolosi per mixnet che utilizzano determinati cifrari.

Si noti che l'avversario potrebbe essere qualunque soggetto in gioco, e quindi potrebbe anche controllare un certo numero di server corrotti: per questo può essere utile aumentare il numero di stage per difendersi. Altre soluzioni preventive prevedono l'utilizzo di prove ZK e meccanismi di autenticazione.

Attacchi meno raffinati, ma comunque pericolosi possono essere in ambito pratico di tipo DoS (Denial of Service) impedendo le comunicazioni di rete fra i vari stage.

2.1.4 Ambiti di Utilizzo delle Mixnets

Descrivendo i requisiti di sicurezza per le mixnets, abbiamo accennato a specifici contesti in cui vengono utilizzati: uno dei più citati è sicuramente quello del voto elettronico o *e-voting*. Considerando lo scenario di un'elezione gli elementi del batch di input della mixnet corrispondono ai voti in forma digitale, i quali vengono permutati e resi anonimi grazie alle trasformazioni crittografiche, per poi essere infine decrittati e conteggiati.

Le proprietà di verificabilità delle mixnets consentono prove pubbliche di sicurezza da parte di terze parti per controllare eventuali tentativi di violazione dell'integrità e dell'anonimità dei voti o di alterazione del conteggio.

Un'altro ambito in cui le mixnets sono impiegate è quello delle telecomunicazioni sicure via web per la trasmissione di informazioni sensibili, per esempio con sistemi di mailing anonimi, in particolare in un contesto come quello di Internet.

Anche per le trasmissioni wireless le mixnets vengono utilizzate per gestire comunicazioni con reti mobili (ad esempio GSM) garantendo la segretezza delle informazioni di localizzazione.

2.2 Classificazione delle Mixnets

Nel corso degli anni, un gran numero di mixnet differenti sono state elaborate, proposte e comparate. Sebbene sia possibile classificare le mixnets attraverso molti criteri diversi, ad esempio secondo la topologia o l'anonimità garantita, data la quantità di varianti esistenti nessuno di essi riesce da solo ad effettuare una categorizzazione sufficientemente precisa.

Per questo adotteremo un sistema di classificazione 'misto' che considera più criteri, sulla base di quello elaborato da Sampigethaya [5].
I criteri utilizzati sono:

- Il tipo di operazione crittografica compiuta negli stage della mixnet: **Decryption**, **Re-encryption** e **Hybrid**;
- I cifrari utilizzati nelle operazione crittografiche: ElGamal o RSA;
- I meccanismi di verificabilità utilizzati.

L'ultimo criterio è il più importante per la classificazione che utilizziamo e merita una discussione preliminare più approfondita. Come già accennato,

la verificabilità per le mixnets è basata su prove interattive che vengono effettuate sugli output dei singoli stage o dell'intera mixnet. Possiamo individuare tre sotto-requisiti necessari per testare la mixnet, in riferimento alla verificabilità di alcuni requisiti sopra descritti:

- **C1**) Il batch di input è stato trasformato crittograficamente e permutato in maniera corretta (requisito di correttezza);
- **C2**) Gli elementi del batch di input non sono stati corrotti (requisito di integrità);
- **C3**) Non sono stati aggiunti o tolti elementi al batch di input (requisito di integrità).

Su queste basi, definiamo le mixnets come:

- **Sender Verifiable**, se rispetta solo il sotto-requisito C2;
- **Stage Verifiable**, se vengono rispettati almeno C1 e C3;
- **Universally Verifiable**, se sono soddisfatti C1, C2 e C3 (vedremo come in realtà le mixnets UV debbano soddisfare anche altri requisiti importanti di robustezza);
- **Conditionally Universally Verifiable**, se viene soddisfatto almeno uno dei sottorequisiti dati.

In Fig. 2.3 abbiamo una rappresentazione grafica esplicativa di questa parziale classificazione.

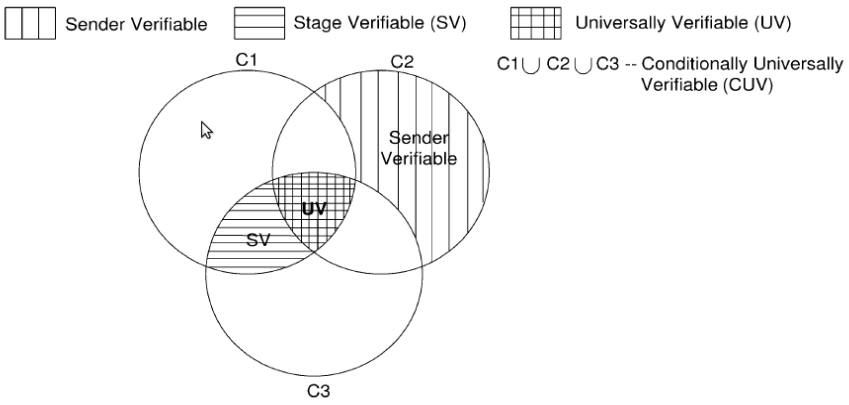

Fig. 2.3: Diagramma di Venn - Classificazione delle mixnets in base a verificabilità

Considerando tutti i criteri complessivamente, possiamo dare una rappresentazione delle categorie di mixnet nel diagramma in Fig. 2.4.

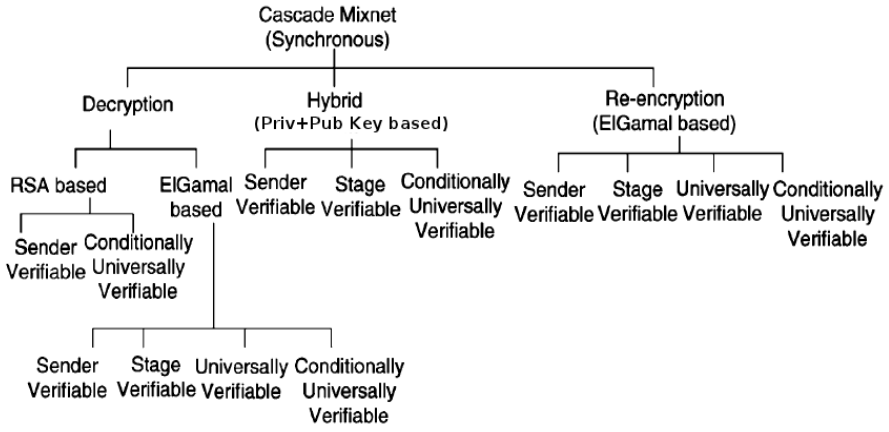

Fig. 2.4: Diagramma di classificazione delle mixnets con multipli criteri

2.2.1 Notazioni

Utilizzeremo in questa parte del paper una serie di notazioni comuni per tutte le mixnets, al fine di facilitarne la comparazione e la comprensione del funzionamento.

Ad ogni stage di una mixnet corrisponde un server M_i con $i = 1, .., l$ dove l rappresenta il numero di stage della mixnet. In una prima fase, le chiavi pubbliche pk_i e private sk_i vengono generate, un paio (pk_i, sk_i) per ogni mix server: mentre le chiavi pubbliche sono visibili a chiunque, le chiavi private devono essere mantenute segrete.

Gli elementi in input saranno m_j con $j = 1, ..., N$ dove N è il numero totale di elementi. Ognuno di essi subisce una qualche trasformazione crittografica iniziale, differente a seconda del tipo di mixnet, generando il batch di input composto dai crittogrammi $c_{0,j}$ per il primo *mix server*. Ogni mix server a partire dal primo provvederà allo stesso modo ad effettuare operazioni crittografiche ed a permutare gli input (trasformazione combinata MCP_i), restituendo infine gli output:

$$\forall i \in [1, l], \forall j \in [1, N] c_{i,\pi_i(j)} = MCP_i(c_{i-1,j})$$

dove $\pi_i : N \to N$ è la permutazione applicata dal server M_i.

In Fig. 2.5 possiamo osservare un'illustrazione grafica di questa impostazione.

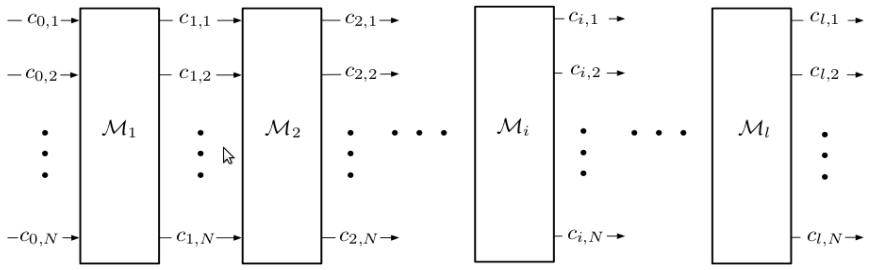

Fig. 2.5: Mixnet con N input ed l server

Quando un mix server M_i fornisce una prova di correttezza utilizza solitamente prove ZK di conoscenza di π_i e dei fattori $r_{i,j}$ usati per randomizzare i crittogrammi. Esistono però alcune tecniche alternative che prevedono verifiche sui mix server di segreti differenti, come ad esempio le chiavi private sk_i.

Nelle sezioni seguenti verranno presentate alcune delle numerose proposte di mix networks elaborate negli anni seguendo l'ordinamento presentato in precedenza e con specifici riferimenti al lavoro di Adida [6].

2.3 Decryption Mixnets

2.3.1 Mixnet di Chaum

La prima mixnet ideata da Chaum nel 1981 appartiene a questa categoria e per questo le decryption mixnet sono spesso denominate anche 'chaumian mixnets'. Esse prevedono l'utilizzo di cifrari a chiave pubblica, RSA in questo caso.

Le chiavi pubbliche pk_i, una per ogni mix server M_i, sono utilizzate all'inizio del protocollo per crittare tutti gli input. Considerati anche i *padding* $r_{i,j}$ per randomizzare i crittogrammi, ogni elemento in input m_j avrà la

seguente forma:

$$c_{0,j} = Enc_{pk_1}(r_{1,j}, Enc_{pk_2}(r_{2,j}, ...Enc_{pk_l}(r_{l,j}, m_j)...))$$

Dato ognuno di questi *onion* in input al primo server della mixnet, ad ogni stage il server M_i lo potrà decrittare con la propria chiave privata sk_i, eliminando lo 'strato' più esterno dell'onion e restituendo in output un nuovo crittogramma privato del padding $r_{i,j}$. L'output finale sarà così l'elemento iniziale m_j. Ad ogni stage inoltre, dopo aver rimosso uno strato dell'onion, si provvede ad effettuare una permutazione π_i. Questa mixnet possiede alcune debolezze, in particolare:

1. **Related Input Attack**: scoperto da Pfitzmann e Pfitzmann nel 1982 [7] è un attacco basato sulle proprietà di omomorfismo di RSA. Se un avversario vuole tracciare un'input $c_{0,j}$, gli basta dare in input un crittogramma $c^* = c_{0,j} * Enc_{pk_1}(f)$ con f piccolo. La relazione algebrica fra $c_{0,j}$ e c^* darà luogo ad una relazione algebrica anche fra gli output crittati e mescolati. Controllando ogni coppia di output presenti, sarà così possibile violare l'anonimità scoprendo gli output corrispondenti a $c_{0,j}$ e c^*;

2. **Sequenza fissa di stage** da seguire per effettuare la decrittazione: ogni strato dell'onion può essere tolto soltanto dal possessore della chiave pubblica pk_i a partire da $i = 1, i = 2...$ ed arrivando a $i = l$;

3. **Dimensione dell'onion** proporzionale al numero di stage, a causa dei padding da concatenare per ogni mix server: la dimensione dell'onion diminuisce proporzionalmente man mano che gli input attraversano i mix server, facilitando il *tracing* da parte di un eventuale avversario.

2.3.2 Decryption Mixnet basata su ElGamal

Per risolvere gli ultimi due problemi evidenziati nella mixnet di Chaum, nel 1993 Park ed altri [8] proposero una decryption mixnet basata sul cifrario a chiave pubblica ElGamal. I parametri utilizzati sono un numero primo p, la fattorizzazione (resa pubblica) di $p - 1$ e g generatore del gruppo ciclico

$\mathbb{Z}_p^*$. Ogni mix server M_i genera una chiave segreta:

$$sk_i = x_i \xleftarrow{\$} \mathbb{Z}_{p-1}^*$$

ed una chiave pubblica:

$$pk_i = y_i = g^{x_i} \, mod \, p_i$$

Rappresenteremo la crittazione di m_j (con r padding random) come:

$$c = Enc_{pk}(m_j; r) = (\alpha, \beta) = (g^r, m_j * y^r)$$

Come in precedenza per RSA, viene effettuata una sola crittazione iniziale degli input, utilizzando come chiave pubblica l'unione di tutte le pk_i:

$$PK = \prod_{i=1}^{l} pk_i = g^{\sum_{i=1}^{l} x_i}$$

Analogamente, ogni mix server M_i potrà effettuare la permutazione degli input e decrittare parzialmente ogni crittogramma:

$$PartialDec_{sk_i}(Enc_{PK}(m_j, r)) =$$

$$= (g^r g^{r_{i,j}} \, , \, m_j \, PK^r \, (g^r)^{-sk_i} \, (\prod_{a \neq i} g^{sk_a})^{r_j})$$

$$= (g^r g^{r_{i,j}} \, , \, m_j \, (g^{\sum_{a \neq i} sk_a r} g^{sk_i r}) \, (g^{-sk_i r}) \, (\prod_{a \neq i} g^{sk_a r_j}))$$

$$= (g^{r + r_{i,j}} \, , \, m_j \, (\prod_{a \neq i} g^{sk_a (r + r_j)}))$$

Quando rimarrà un solo mix server a dover decrittare gli input, otterrà per ognuno di essi $(g^{\sum_{i=1}^{n} r_{i,j}}, m_j)$.

Oltre alla decrittazione parziale, viene utilizzato ad ogni stage un padding random r_i per ogni elemento in input, in modo da rerandomizzare i ciphertext, allo stesso modo in cui operano le reencryption mixnet che saranno illustrate in seguito.

Si noti come l'ordine di decrittazione non abbia importanza e come la dimensione del crittogramma sia sempre la stessa, dato che la trasformazione è soltanto algebrica e non prevede la concatenazione di padding come per RSA.

2.4 Reencryption Mixnet

Come già visto, nel 1993 Park ed altri proposero la prima decryption mixnet basata su ElGamal. Oltre ad essa, elaborarono anche una versione differente che sarebbe stata la prima *reencryption mixnet*: con essa ogni mix server effettua una rerandomizzazione dei crittogrammi utilizzando ogni volta nuovi valori di randomizzazione, i quali vengono combinati algebricamente, anziché concatenati come nelle chaumian mixnet che utilizzano RSA.

Anche in questo caso viene impiegato ElGamal ed il setup iniziale è identico alla variante con decrittazione: vengono generate le chiavi pubbliche pk_i e private sk_i, nonché la chiave pubblica PK della mixnet, data dall'unione di tutte le chiavi pubbliche di ogni stage. Utilizzando PK ogni plaintext m_j viene crittato e dato in input ad M_1.

A questo punto però, anziché decrittare parzialmente ogni crittogramma, il server M_i provvederà a re-crittare il crittogramma con un nuovo padding random $r_{i,j}$ e PK:

$$c_{i,j} = RE_{PK}(c_{i-1}; r_{i,j})$$

L'operazione di reencryption sfrutta le proprietà omomorfiche di ElGamal:

$$RE_{pk}(c; r') = (g^r * g^{r'}, m * y^r * y^{r'}) = Enc_{pk}(m; r + r')$$

Ogni mix server effettua anche una permutazione casuale dei crittogrammi appena rerandomizzati. Quando tutti i mix server hanno terminato la fase di reencryption, l'ultimo batch di output di M_l viene decrittato congiuntamente da tutti i mix server, utilizzando uno schema (t, n)-*threshold* per garantire fault-tolerance. Questa proprietà non può essere invece garantita

dalla variante originale della mixnet con decryption, per la quale un solo stage corrotto può risultare fatale.

Questa reencryption mixnet ha in comune con la variante con decryption alcune debolezze significative, dovute proprio al cifrario a chiave pubblica. Anzitutto, ElGamal non è *semantically secure* (o CPA-sicuro) nella versione descritta in precedenza, dato che utilizza l'intero gruppo $\mathbb{Z}_p^*$: in ogni crittogramma $c = (g^r, m_j * y^r)$, sia g^r che $m_j * y^r$ appartengono a $\mathbb{Z}_p^*$ ed è possibile, anche con un attacco passivo (*semantic security attack*) individuare correlazioni comparando gli input del primo mix server con gli output prima della decrittazione finale: esiste quindi una probabilità non trascurabile di poter dedurre a quali sottogruppi gli output dei mix server g^r e $m_j * y^r$ appartengono. Ciò è possibile in quanto la fattorizzazione di $p-1$ è pubblica e quindi è facile vedere se un elemento a appartiene ad un certo sottogruppo G_f testando se $a^{p-1}/f = 1 \, mod \, p$ con f fattore primo di $p-1$.

Per fortuna è possibile rendere ElGamal CPA-sicuro generando p come numero primo sufficientemente grande ed utilizzando un altro grande numero primo q tale che q divida $(p-1)$; tutti i crittogrammi (ed i messaggi m_j) dovranno appartenere al sottogruppo di ordine q che appartiene a $\mathbb{Z}_p^*$.

Questo fix è però inutile per quanto concerne un altro tipo di attacco, stavolta attivo, simile all'input related attack già visto per le chaumian mixnet. A causa delle proprietà di omomorfismo di ElGamal, dato un crittogramma $c = (\alpha, \beta)$ dal batch di input iniziale, un sender avversario $\mathcal{A}$ può creare un input che ha una relazione algebrica con esso, $c^* = (\alpha^e, \beta^e)$ ed aggiungerlo al batch. La relazione algebrica verrà mantenuta anche dopo l'elaborazione attraverso i mix server, per cui non si dovrà fare altro che controllare ogni coppia di plaintext (m_0, m_1) in output alla mixnet per vedere se sussiste la relazione $m_0^e = m_1$ e, con alta probabilità, solo i plaintext di c e c^* l'avranno.

Una contromisura specifica è quella di dare ad ElGamal la proprietà di *Non-Malleability*, facendo in modo che non sia possibile produrre input algebricamente relazionati ad altri: per fare questo vengono utilizzate tecniche per la sicurezza contro attacchi *chosen ciphertext*.

2.4.1 Universal Reencryption Mixnet

Verrà descritta ora una semplice variante di reencryption mixnet, elaborata da Golle ed altri [9], che consente di non far conoscere ai mix server la chiave pubblica con la quale operano, in modo che essi non debbano obbligatoria-

mente partecipare ai processi di generazione, distribuzione e gestione delle chiavi.

Il sistema prevede che durante la fase di crittazione iniziale si generino input per il primo mix server formati in questo modo:

$$c = (Enc_{pk}(m), Enc_{pk}(1)) = (\ (g^r, m * y^r)\ ,\ (g^s, y^s)\ )$$

dove r, s sono fattori di randomizzazione. A questo punto, ogni mix server M_i riceverà in input ogni coppia e potrà effettuare la reencryption, utilizzando una forma crittata della pk e generandone una differente con un nuovo padding:

$$UnivRE_{pk}(c; r', r'') = (Enc_{pk}(m) * Enc_{pk}(1)^{r'}, Enc_{pk}(1)^{r''})$$

dove $r^{'}$ è il padding per randomizzare i crittogrammi, e r'' il padding per randomizzare la crittazione della chiave pubblica $Enc_{pk}(1)$ per il mix server successivo.

Golle propone l'utilizzo di ElGamal semantically secure, come descritto in precedenza, scegliendo r', r'' appartenenti ad un sottogruppo di ordine q di $\mathbb{Z}_p^*$. L'assunzione DDH per ElGamal assicura che da $Enc_{pk}(1)$ non sia deducibile pk.

2.5 Hybrid Mixnet

Abbiamo visto come per le decryption mixnet si utilizzino cifrari come RSA o ElGamal, a chiave pubblica. In quanto tali essi presentano due problematiche:

- Lo spazio dei messaggi in chiaro e quello dei crittogrammi è assai ristretto: ad esempio, nel caso di ElGamal, poche centinaia di *bits*;
- Le operazioni crittografiche con chiave pubblica sono circa 10^3 volte più computazionalmente onerose rispetto a quelle che impiegano chiave simmetrica.

Nella pratica quindi, si è pensato di utilizzare sistemi in cui si potessero combinare le proprietà di sicurezza dei cifrari a chiave pubblica con l'efficienza e

la flessibilità di quelli a chiave simmetrica: in questa maniera sarebbe possibile dare in input ad una mixnet crittogrammi di dimensione arbitraria.

Nei sistemi ibridi si utilizzano generalmente onion formati dalla chiave simmetrica crittata con la chiave pubblica ed una crittazione, con la chiave simmetrica stessa, dell'onion da inviare allo stage successivo. Ad esempio, il mix server M_j produrrà il seguente onion:

$$Onion_j = (Enc_{K_j}(k_j)\,,\, Enc_{k_j}(Onion_{j+1}))$$

dove K_j è la chiave pubblica del mix server j-esimo e k_j una chiave privata generata dal sender a partire da quella pubblica. Come si può vedere l'unica operazione di decrittazione che fa uso di K_j è da effettuare sulla chiave privata che è di dimensione non crescente, mentre per l'onion si utilizza la chiave privata.

2.5.1 Optimally Robust Hybrid Mix Network

Una proposta interessante di mixnet ibrida *ElGamal based* è la 'Optimally Robust Hybrid Mix Network' di Jakobsson e Juels [10]: viene definita come *Optimally Robust* in quanto garantisce il requisito di robustezza nel caso in cui meno di metà dei mix server siano corrotti.

Sono impiegate due differenti simmetriche, una per crittare i messaggi come visto nell'impostazione generale ed una per generare un MAC (*Message Authentication Code*) usato per verificare l'integrità dei crittogrammi. Abbiamo tre fasi distinte del protocollo:

Setup: ogni mix server M_i sceglie tre chiavi private α_i, β_i e $\gamma_i \in G$ di ordine q, dove $G \subset \mathbb{Z}_p^*$ e genera tre chiavi pubbliche calcolando $Y_i = Y_{i-1}^{\alpha_i}$, $K_i = Y_{i-1}^{\beta_i}$ e $\mathbb{Z}_i = Y_{i-1}^{\gamma_i}$, con valore di *bootstrap* $Y_0 = g$, generatore del gruppo ciclico a cui appartengono le chiavi pubbliche e $i = 1, ..., l$.

Crittazione Iniziale: il sender sceglie un segreto ρ e calcola due chiavi simmetriche:

$$\begin{cases} k_i = K_i^{\rho} & con \quad 0 \leq i \leq l+1 \\ z_i = \mathbb{Z}_i^{\rho} & con \quad 0 \leq i \leq l+1 \end{cases}$$

Viene inoltre calcolata una chiave compressa di *scheduling* $y_0 = Y_0^{\rho}$. Ogni messaggio m viene crittato ad ogni stage (in ordine inverso) con k_i e ne viene

calcolato il MAC con z_i. Ogni mix server M_i calcola:

$$c_i = Enc_{k_{i+1}}(c_{i+1}, \mu_{i+1}) \quad con \quad 0 \leq i \leq l-1$$

$$z_i = MAC_{z_{i+1}}(c_i) \quad con \quad 0 \leq i \leq l$$

Alla fine del processo otteniamo un crittogramma $C = (y_0, c_0, \mu_0)$ dove c_0 è la crittazione con chiave simmetrica di m, μ_0 è il suo MAC e y_0 la sua chiave compressa di scheduling. Un batch di N input C viene perciò passato al primo mix server M_0:

$$(y_0^j, c_0^j, \mu_0^j)_{j=1}^N$$

Mixing: una volta ricevuto il batch di input ogni server M_i usa le sue tre chiavi private α_i, β_i e γ_i e la chiave compressa di scheduling y_{i-1} ricevuta da M_{i-1} per generare le due chiavi simmetriche k_i e z_i, nonchè la nuova chiave compressa di scheduling y_i, senza conoscere direttamente il segreto ρ del sender:

$$\begin{cases} k_i = y_{i-1}^{\gamma_i} \\ z_i = y_{i-1}^{\beta_i} \\ y_i = y_{i-1}^{\alpha_i} \end{cases}$$

Quando M_i riceve un crittogramma in input ne verifica l'integrità controllando che il MAC incluso sia corretto, ricalcolandolo in questo modo:

$$z_{i-1} = MAC_{z_i}(c_{i-1})$$

Se l'operazione ha successo, allora si utilizza la chiave simmetrica k_i per decrittare uno strato del crittogramma (come per le decryption mixnet):

$$D_{k_i}(c_{i-1}) = (c_i, \mu_i)$$

Ovviamente, una volta giunti a c_l avremo soltanto la crittazione del messaggio m. Dopo la decrittazione, il mix server non fa altro che permutare casualmente i crittogrammi e restituirli in output, fornendo infine una prova

di correttezza delle chiavi compresse di scheduling y_i.

Mentre quest'ultima proposta non è affetta dagli svantaggi propri delle decryption mixnet basate su RSA (onion) già descritte, altre mixnets ibride antecedenti lo sono, in quanto basate sull'uso di onions che diminuiscono di dimensione mentre attraversano i vari stage.

2.6 Meccanismi di Verificabilità per Mixnets

Come abbiamo discusso in precedenza, esistono diversi tipi di mixnets in base ai criteri di verificabilità C1, C2, C3 prestabiliti. Ora vedremo nel dettaglio quali caratteristiche contraddistinguono queste tipologie.

2.6.1 Mixnet Sender Verifiable

Le mixnets *sender verifiable* sono in grado soltanto di rilevare manipolazioni dei messaggi volte a violare i requisiti di integrità (requisito C2). Anche la prima mixnet di Chaum ha una variante che permette di garantire *sender verifiability*.

In sostanza, ad ogni crittogramma di input per la mixnet viene aggiunto un *checksum* in forma di padding composto da zeri:

$$Enc_{pk}(sender - pk, (D_{sender-sk}(0 - padded\, m)), r)$$

dove $sender - pk$ e $sender - sk$ sono rispettivamente la chiave pubblica e privata del sender, mentre $0 - padded\, m$ è il messaggio iniziale con incluso un padding di alcuni zeri (pubblico). Al termine del mixing, avremo a disposizione l'input decrittato, $sender - pk, (D_{sender-sk}(0 - padded\, m))$; a questo punto, il sender come chiunque altro potrà usare $sender - pk$ per decrittare $D_{sender-sk}(0 - padded\, m)$ (impiegando lo schema crittografico in maniera inversa) e verificare l'integrità di m: qualunque modifica al messaggio comporterebbe infatti anche una modifica del messaggio con il padding. Si noti come $D_{sender-sk}(0 - padded\, m)$ possa essere preparato soltanto dal sender, il quale conosce $sender - sk$.

Tale meccanismo prevede che ogni sender debba verificare il proprio messaggio, altrimenti è possibile per qualche server corrotto sostituire quell'input senza essere identificato, violando il requisito C3. Per risolvere questo problema necessitiamo di mixnet *stage verifiable*.

2.6.2 Stage Verifiable Reencryption Mixnet

Nelle cosiddette Stage Verifiable mixnets le verifiche sono effettuate dagli stessi mix server, senza bisogno di verifiche da parte dei sender: bastano prove ZK ed altri sistemi crittografici.

Nelle mixnets proposte da Jakobsson in [11], [12] vengono utilizzate diverse importanti tecniche, vediamone alcune:

1. **Duplicazione degli input batch**: per quanto oneroso computazionalmente, creare n copie di ogni input batch e fornirle ad ogni mix server rende complicato corrompere un input, dato che ogni input batch subisce permutazioni e crittazioni diverse. Uno stage corrotto può ad esempio manipolare il seondo elemento del primo input batch che riceve, ma con scarsa probabilità riuscirà a manipolare lo stesso elemento in tutti gli altri $n-1$ batch. Ogni tentativo viene comunque rilevato rivelando i segreti e comparando i batch in output, come vedremo.

2. **Ripetizione della fase di mixing**: come si può notare intuitivamente, duplicare i batch di input mantiene l'integrità contro ogni stage corrotto, ma non con il primo: dato che esso inizia la fase di mixing può trovare lo stesso input in tutti i batch e manipolarlo, dato che gli input batch sono identici. Per evitare ciò è possibile applicare il principio di *repetition robustness* della Practical Mix di Jakobsson, applicando molteplici fasi di mixing. In questo modo, se il primo stage corrompe degli input, alla fase di mixing successivo riceverà l'output batch della fase precedente e non riuscirà a localizzare gli input modificati all'inizio. La corruzione degli input potrà essere rilevata come descritto in seguito.

3. **Rivelazione dei segreti e comparazione dei batch di output**: nelle reencryption mixnet si utilizza lo schema ElGamal, ed è grazie alle sue proprietà di omomorfismo che possiamo rilevare stage compromessi. Poniamo che uno stage M_i corrotto modifichi un input $Enc_{pk}(m;r)$, utilizzando mm' al posto di m e fornisca quindi in output:

$$Enc_{pk}(mm';r+r') = (g^{r+r'}, mm' * pk^{r+r'}) = (g^{r+r'}, mm' * g^{(r+r')sk})$$

Sappiamo che in ElGamal $m' \in \mathbb{Z}_p^*$ si può esprimere come g^x per qualche x, ottenendo:

$$(g^{r+r'}, mg^x * g^{(r+r')sk}) = (g^{r+r'}, m * g^{(r+r')sk+x})$$

Ora basta rivelare r' (r è dato in input con $Enc_{pk}(m; r)$) per individuare il fattore g^x e trovare M_i corrotto. Rivelare r' viola però i requisiti di anonimità della mixnet: per questo la rivelazione viene effettuata soltanto a partire dalla seconda fase di mixing in poi.

4. **Tracing degli stage corrotti**: quando vengono rilevate anomalie in qualche stage, viene effettuato un tracing inverso: ogni stage rivela i propri segreti e le sue operazioni sono controllate dagli stage rimanenti, evitando di decrittare però gli output per preservare l'anonimità.

5. **Prove ZK per verifica dell'integrità**: per soddisfare il requisito C3, ovvero per verificare se elementi in input ad uno stage vengano aggiunti o eliminati, è possibile impiegare una prova ZK al termine della fase di mixing e controllare se il prodotto degli input della mixnet corrisponde a quello degli output, utilizzando opportuni esponenti di *unblinding*. Una prova ZK può essere richiesta anche prima della fase di mixing, per verificare la conoscenza di parte della chiave privata della mixnet. Ogni prova fornita da un server è verificata dai rimanenti stage: se qualche stage risulta corrotto, viene avviato un sotto-protocollo qui descritto.

6. **Protocollo di reazione ad uno stage corrotto**: se uno stage viene trovato corrotto prima della fase di mixing, esso viene rimosso o simulato dagli altri stage in collaborazione; se viene rilevato dopo l'inizio della fase di mixing, esso viene rimosso ed il processo dell'intera mixnet riavviato.

Tra la *Practical Mixnet* e la *Flash mixnet* di Jakobsson esistono alcune differenze: ad esempio, nella practical mix, un solo stage corrotto può sfruttare le proprietà omomorfiche di ElGamal e modificare tutti gli input senza essere scoperto; per farlo basta che mantenga il prodotto degli input in modo da superare la prova ZK del prodotto aggregato. Nella Flash Mixnet ciò non è possibile perchè si utilizza la tecnica dei *dummy inputs*. In entrambe le

proposte però, una coalizione corrotta di tutti gli stage può violare entrambe le mixnets, che non sono quindi universally verifiable.

Desmedt e Kurosawa proposero una versione della practical mix [13] che rispetta tutti i requisiti C1, C2 e C3, esattamente come le mixnets UV. Essa prevede che ogni stage sia verificato da altri $t-1$ stage, rivelando i segreti ad essi in modo da poterne controllare gli output. In caso di anomalie al server M_i, l'ouput di M_{i-1} passa direttamente in input a M_{i+1}. In una mixnet con t stage, avremo un totale di $(t-1)^2$ stage. La fault tolerance rimane comunque limitata a $t-1$ server corrotti: se un verifier in ogni stage è corrotto, questi t verifier possono rivelare gli esponenti di randomizzazione segreti e rompere l'anonimità della mixnet.

2.6.3 Stage Verifiable Hybrid Mixnet

Descriviamo qui le tecniche usate per la verificabilità nella 'Optimally Robust Hybrid Mix Network' già descritta:

1. **Generazione della chiave simmetrica da quella pubblica**: la chiave privata k_i di ogni stage è generata a partire dalla chiave pubblica K_i e da y_{i-1} che funge da *key generator* o *keygen*, ottenendo $k_i = y_{i-1}^{\gamma_i}$. Il keygen viene aggiornato ad ogni stage con l'esponenziazione, usando la chiave privata α: $(y_i = y_{i-1}^{\alpha_i})$. Vedremo come questa operazione sia coinvolta in prove ZK per la verifica dei mix server;

2. **Message Authentication Code**: come abbiamo visto, il MAC incluso nell'onion permette ad uno stage M_i di verificare l'integrità del batch ricevuto da M_{i-1};

3. **Generazione delle prove, Verifica e Reazione agli stage corrotti**: ogni server M_i deve dare una prova di conoscenza della propria chiave privata α_i usata per esponenziare il keygen, una prova ZK computazionalmente efficiente viene usata a questo scopo. Questa prova, assieme al controllo sul MAC, consente di rilevare stage corrotti. Nel caso ciò avvenga per il server M_i, un gruppo soglia di t stage diversi genera la chiave privata dello stage corrotto e controlla le operazioni di mixing: se si verifica l'effettiva corruzione, M_i viene sostituito dagli altri t stage. Se invece le operazioni di mixing sono corrette, allora la corruzione può riguardare:

- lo stage M_{i+1} che afferma di avere rilevato M_i come corrotto;
- uno o più input della mixnet: inizia una fase di tracing inverso per trovare gli input errati e cancellarli, seguita da una ripetizione della fase di mixing.

4. **Simulazione dell'ultimo stage**: l'ultimo stage non possiede un mix server successivo M_{l+1} per fare verifiche sull'integrità degli output di M_l. Lo stage M_{l+1} viene perciò simulato dagli l stage in collaborazione.

Queste misure utilizzate per mixnet ibride consentono di soddisfare tutti i requisiti di verificabilità C1, C2 e C3, anche se permangono rischi riguardanti la rivelazione dei segreti in presenza di errori negli input, con i quali un avversario può lanciare un *anonimity attack*.

2.6.4 Limiti delle Mixnets Stage Verifiable

A prescindere dai requisiti di correttezza, che per le mixnets SV possono anche essere tutti rispettati, un importante loro svantaggio è legato alla fault tolerance. Dato che la responsabilità delle verifiche di correttezza è assegnata ai singoli stage, una coalizione di tutti i mix server può corrompere gli input senza essere rilevata dal protocollo. Per risolvere questo problema sono state progettate le mixnets UV, Universally Verifiable.

2.6.5 Mixnet Universally Verifiable

L'idea alla base di questo tipo di mixnet è quella per la quale ogni stage debba provare che ogni elemento nel proprio batch di output corrisponda ad un solo elemento nel proprio batch di input, senza svelare la relazione fra essi: prove ZK sono utilizzate per questo obiettivo. Per quanto concerne invece la reazione agli stage compromessi, il comportamento è lo stesso descritto per le mixnets SV.

La denominazione Universally Verifiable deriva dal fatto che la verifica delle prove di correttezza fornite dagli stage deve potere essere eseguita da qualunque osservatore esterno.

La prima mixnet UV è stata elaborata nel 1995 da Sako e Kilian basandosi sulla reencryption/decryption mixnet di Park. In questa proposta ad ogni stage occorre che il mix server fornisca il proprio batch di output nonché le prove di correttezza di decrittazione, reencryption e permutazione.

Per provare che reencryption con valori di randomizzazione r_j e permutazione π_j sono corrette, ogni mixnet effettua aggiuntivamente una nuova ed indipendente fase di permutazione λ_j e reencryption con valori di randomizzazione t_j. Un *verifier* richiede poi al mix server M_i di mostrargli una tra le seguenti opzioni:

- La differenza tra la prima e seconda fase di reencryption/permutazione $(\lambda_j * \pi_j^{-1}, (r_j - t_i))$, per potere scoprire come ottenere la prima permutazione π_j ed i valori di randomizzazione r_j a partire dai risultati dell'ultima fase;

- I valori di (λ_j,t_j) per verificare che la seconda fase sia avvenuta correttamente.

In questa versione però, l'anonimità può essere violata, come dimostrato dai successivi attacchi di Michels-Horster. Non esistono inoltre contromisure e meccanismi di *recovery* relativamente a stage corrotti.

Una versione più elaborata di questa mixnet UV è stata proposta da Abe nel 1998 [14] e permette di diminuire la quantità di lavoro richiesta al verifier e di renderla indipendente dal numero di stage della mixnet. Si prevede che tutti i mix server debbano generare una prova di correttezza collettiva che possa essere controllata da un qualsiasi verifier. In questo modo, anche se tutti i mix server sono corrotti, è possibile rilevare la violazione e garantire pienamente la proprietà di fault tolerance.

Abbiamo una reencryption mixnet basata su ElGamal in cui ogni M_i possiede una parte della chiave segreta sk secondo uno schema threshold: per provare la correttezza delle proprie operazioni, ogni mix server utilizza una seconda fase di reencryption e permutazione, in questo caso però ognuna di queste fasi è basata sulla seconda fase dello stage precedente, creando una sorta di 'catena' di prove aggiuntive. In questo modo il verifier deve effettuare un solo controllo per l'intera mixnet, a scelta tra:

- La differenza *mixnet-wide* tra prima e seconda fase di reencryption/-permutazione;

- I valori *mixnet-wide* della seconda fase di reencryption/permutazione.

Entrambi possono essere calcolati in sequenza dai mix server, senza oneri da parte del verifier.

Per quanto riguarda le decryption mixnet viene utilizzata una versione

di ElGamal threshold in cui i mix server, in numero sufficiente a superare la soglia predefinita, provvedono insieme a fornire un *decryption factor* $\gamma_j = \alpha^x$ per ogni output della mixnet $c_{l,j} = (\alpha_j, \beta_j)$. A questo fine vengono combinate porzioni algebriche di γ_j appartenenti ad ogni M_i, ossia $\gamma_{j,i} = \alpha_j^{x_i L_i}$ dove x_i è una parte (algebrica) della chiave segreta x della mixnet e L_i il fattore di interpolazione di Lagrange dello stage. I server producono inoltre prove di conoscenza della correttezza della produzione dei decryption factors.

In seguito sono state elaborate mixnets UV basate su permutation network come quelle di Abe [15] e di Jakobsson e Juels [16] che fanno uso di assunzioni specifiche al sistema crittografico da loro utilizzato (ad es. ElGamal): in questo modo è possibile avere prove computazionalmente più efficienti.

Ricapitolando, a differenza delle SV mixnet, le mixnets UV garantiscono verificabilità esterna e, in presenza di stage corrotti, non è richiesto un riavvio del protocollo.

2.6.6 Mixnet Conditionally Universally Verifiable

Le mixnets CUV sono così definite a causa del fatto che utilizzano meccanismi simili a quelli delle mixnets UV, ma è possibile rilevare anomalie soltanto con un certo grado di probabilità: questo porta le mixnets CUV a soddisfare i requisiti di verificabilità C1, C2 e C3 solo probabilisticamente.

Nel 2002 Jakobsson, Juels e Rivest [17] idearono RPC, un sistema generico per effettuare prove di correttezza indipendentemente dal sistema crittografico usato e quindi utilizzabile come sotto-protocollo al termine della fase di mixing in qualunque mixnet esistente.

Il concetto di base è immediato: ogni mix server rivela una parte delle corrispondenze tra input ed output. Dato in output un crittogramma $c_{i,j}$ a caso, M_j rivela $(\pi_i(j), r_{i,j})$, ovvero la permutazione effettuata per associarlo all'elemento corrispondente nel batch di input ed il valore di rerandomizzazione usato per crittarlo; nel caso in cui la mixnet non usi valori di rerandomizzazione (ad esempio una decryption mixnet), viene rivelata solo la permutazione $\pi_i(j)$. La scelta di quali crittogrammi in output controllare è a discrezione del verifier esterno.

Come si può notare, con le impostazioni attuali, sarebbe semplice per un verifier violare l'anonimità: gli basterebbe, ad ogni stage, rivelare i segreti di un crittogramma in output corrispondente a quello rivelato in precedenza,

percorrendo tutte le permutazioni svolte e svelando la corrispondenza tra l'output finale della mixnet e l'input originale.

Per evitare questo, i mix server vengono sempre verificati a coppie: ogni verifier richiede di verificare una serie di crittogrammi $c_{i,j}$ a sua scelta e per metà di essi M_i rivela gli input corrispondenti $c_{\pi^{-1}(i),j-1}$, mentre per l'altra metà è M_{j+1}, lo stage successivo, a rivelare invece gli output $c_{\pi(i),j+1}$ corrispondenti. Vediamo in Fig. 2.6 un'illustrazione grafica del funzionamento. Rivelare segreti di alcuni elementi comporta una parziale perdità

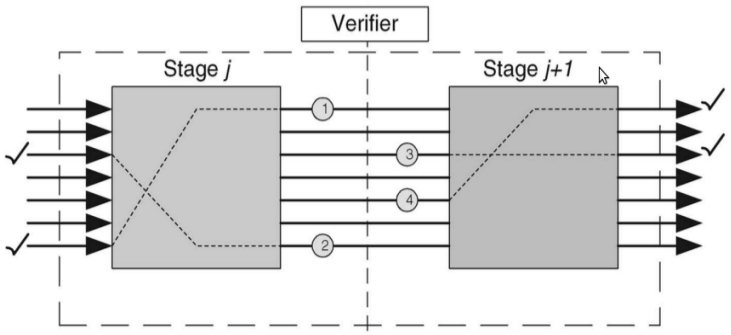

Fig. 2.6: Randomized Partial Checking - Il verifier sceglie crittogrammi 1,2,3,4: per 1,2 lo stage j svela gli input corrispondenti; per 3,4 lo stage j+1 rivela le corrispondenze nel proprio batch di output

di anonimità, compensata però spesso dalle grandi dimensioni del batch di input. Per quanto concerne la fault tolerance è sufficiente che una coppia di mix server consecutivi sia onesta per garantire l'anonimità. L'efficienza computazionale è piuttosto buona, dato che non sono necessarie prove ZK.

Un grosso svantaggio rispetto alle mixnets UV è invece dato dal fatto che non si ha la sicurezza di scoprire mix server corrotti: i crittogrammi rilevati potrebbero non comprendere quelli manipolati o aggiunti. La probabilità di non rilevare elementi corrotti rimane comunque estremamente bassa per quanto riguarda violazioni d'integrità in quantità significative.

Sempre nel 2002, Golle ed altri [18] idearono una tecnica CUV per reencryption mixnet che utilizzano ElGamal. Nella mixnet che fa uso di optimistic verification sono usati due livelli di crittazione. Inizialmente i plaintext vengono crittati come di consueto:

$$c = Enc_{pk}(m_j; r) = (\alpha, \beta) = (g^r, m_j * y^r)$$

successivamente c e le sue componenti vengono crittate, sempre con ElGamal, ad un secondo livello, impiegando anche l'hashing per creare un *checksum*:

$$d = (d_1, d_2, d_3) = (Enc(\alpha; r'), Enc(\beta; s'), Enc(\alpha, \beta); t'))$$

dove r', s', t' sono valori di randomizzazione ed H è una funzione *hash one-way.*

Successivamente vengono compiute le fasi di reencryption e di decrittazione finale (con soglia di mix server prestabilita) ottenendo triple (α, β, γ). Viene fornita inoltre la prova di correttezza (efficiente) delle *aggregate properties*: ogni stage prova che il prodotto degli input è uguale al prodotto degli output.

Un'altra prova ZK di correttezza è usata per dimostrare che ad ogni input corrisponde un elemento in output. Si noti come un avversario potrebbe mantenere il prodotto degli output identico a quello degli input, modificando opportunamente gli output: con il checksum chiunque può verificare se ciò avviene, dato che ogni modifica ad un crittogramma comporta anche una modifica del checksum stesso, ottenendo $\gamma \neq H(\alpha, \beta)$.

Quando una violazione d'integrità viene rilevata è però necessario riavviare l'intero protocollo: per questo motivo le mixnets con optimistic verification, pur rispettando tutti i requisiti C1, C2 e C3, sono classificate come CUV anziché come UV.

Capitolo 3

Prove Game-Based e EasyCrypt

3.1 Origini dell'Approccio Riduzionistico

A partire dal report [19] di Shannon, il quale dimostrò nel 1949 che la nozione di sicurezza perfetta poteva essere raggiunta soltanto utilizzando chiavi simmetriche di dimensione uguale o maggiore a quella dei messaggi da crittare, è risultato chiaro che i sistemi crittografici dovesssero essere studiati come oggetti matematici e seguendo metodologie specifiche e rigorose.

Lo sviluppo della teoria della complessità computazionale ha consentito la possibilità di esprimere le proprietà di sicurezza probabilisticamente: in questo senso il lavoro di Goldwasser e Micali [4] del 1984 definisce un nuovo modello per la dimostrazione delle proprietà di sicurezza per cifrari a chiave pubblica.

In particolare il modello stabilisce i seguenti punti:

- Occorre definire i requisiti di sicurezza del sistema ed il modello dell'avversario, ossia le risorse computazionali (di solito limitate polinomialmente) e le informazioni a sua disposizione, come ad esempio la crittazione di due plaintext a sua scelta e la chiave pubblica: da ciò deriva la definizione del CPA indistinguishability experiment con la possibilità per un avversario $\mathcal{A}$ di accedere ad oracoli;
- Devono essere formalizzate precisamente in senso probabilistico le assunzioni di sicurezza per cui la definizione di *Enc* risulta una funzione *trapdoor* che sia *easy-to-compute*, ma *hard-to-invert*: tali assunzioni si

basano sulla difficoltà pratica di risolvere un problema o una decisione, come nel caso dell'assunzione DDH su cui ElGamal si basa.

- Si descrive formalmente un sistema crittografico a chiave pubblica π come definito nel Capitolo 1, evidenziando come l'algoritmo di crittazione *Enc* debba essere probabilistico affinché π sia polinomialmente sicuro;

- Viene definita formalmente la nozione di sicurezza semantica di un sistema π, per cui un avversario *polinomially bounded* ha la stessa probabilità di dedurre informazioni su un plaintext sia avendone a disposizione il crittogramma, sia non avendolo: si dimostra inoltre che ogni sistema crittografico a chiave pubblica π è semanticamente sicuro.

La sicurezza di un sistema π è quindi dimostrabile a partire da un esperimento, mostrando una riduzione ad una o più assunzioni computazionali: proprio da questa osservazioni prende spunto la strategia game-based che andremo a illustrare.

3.2 Prove Game-Based secondo Shoup

Nell'articolo di Shoup del 2004 [20] viene descritta la tecnica delle sequenze di giochi come una strategia per organizzare dimostrazioni di sicurezza inerenti a sistemi crittografici e poter gestire al meglio la complessità delle prove. Non tutte le prove di sicurezza sono gestibili con sequenze di giochi e comunque si tratta di un metodo organizzativo che esula dalle idee alla base della dimostrazione stessa.

Una importante conseguenza di questo approccio è la possibilità di strutturare le dimostrazioni al fine di renderle più comprensibili, evidenziandone i passi cruciali e rendendone più semplice la verifica: i software per dimostrazioni basate su giochi trovano fondamenti teorici proprio nelle tecniche che stiamo per descrivere.

3.2.1 Definizione di Sequenze di Giochi ed Eventi

Possiamo definire una sequenza di game come segue:

> Una sequenza di giochi $G_0, G_1, ..., G_n$ corrisponde ad una serie di esperimenti in cui G_0 è l'obiettivo di sicurezza per un certo sistema crittografico in riferimento ad un dato avversario, mentre G_n (n costante) rappresenta l'assunzione di sicurezza a cui l'evento di interesse è legata.

Abbiamo poi eventi e probabilità inerenti:

> $E_0, E_i, ..., E_n$ sono una serie di eventi relativi ai giochi con E_0 come evento di interesse nell'obiettivo di sicurezza e $Pr[E_n]$ la *target probability* relativa all'assunzione.

La dimostrazione di sicurezza è ottenuta quindi:

> Verificando che $\forall_{i \in \{0...n\}}\ Pr[E_i] = Pr[E_{i+1}] \pm negl$, dato che n è costante, si dimostra che $Pr[E_0]$ e $Pr[E_n]$ sono *negligibly close*, facendo così coincidere la probabilità dell'evento relativo all'obiettivo di sicurezza con la target probability dell'assunzione specificata.

Ovviamente, se le assunzioni su cui poggia il sistema sono molteplici, avremo una struttura ad albero in cui vi sono diversi percorsi che rappresentano le probabilità di transizione da un gioco ai suoi successori e che terminano con assunzioni differenti.

3.2.2 Transizioni fra Giochi

Le transizioni da un gioco ad un altro possono essere di tre tipi diversi:

- **Transizioni basate su indistinguibilità**: in questa tipologia di transizione, si ha un cambiamento minimale da G_i a G_{i+1}, per esempio la sostituzione di una istruzione deterministica con una random. Data la definizione $\forall_{i \in \{0...n\}}\ \ |Pr[E_i] - Pr[E_{i+1}]| = negl$ è possibile costruire un distinguitore D, il quale prende in input elementi dalle distribuzioni di output di G_i e G_{i+1}: nel primo caso restituisce 1 con probabilità $Pr[E_i]$, nel secondo caso con probabilità $Pr[E_{i+1}]$ e quindi con differenza probabilistica trascurabile. In generale è possibile esprimere un singolo game che, a seconda della distribuzione da cui riceve un input, si può comportare come G_i oppure G_{i+1};

- **Transizioni basate su bad events**: in queste transizioni si hanno comportamenti identici dei giochi G_i e G_{i+1}, fino a quando non occorre un *bad event*, detto anche *failure event* F. Ciò lo possiamo esprimere logicamente come:

$$E_i \wedge \neg F \iff E_{i+1} \wedge \neg F$$

  In questo caso possiamo usare il *Fundamental Lemma*:
  Se A, B, F sono eventi in una stessa distribuzione di probabilità e vale che $E_i \wedge \neg F \iff E_{i+1} \wedge \neg F$ allora $|Pr[A] - Pr[B]| \leq Pr[F]$.
  Dim.:

$$|Pr[A]-Pr[B]| = ||Pr[A]\wedge F|+|Pr[A]\wedge\neg F|-|Pr[B]\wedge F|-|Pr[B]\wedge\neg F||$$

$$= ||Pr[A] \wedge F| - |Pr[B] \wedge F|| \leq Pr[F]$$

  Il teorema afferma che la probabilità che un avversario $\mathcal{A}$ possa distinguere tra due game identici fino all'evento F è al più la probabilità che in G_i o G_{i+1} avvenga un bad event, perciò è sufficiente dimostrare che $Pr[F]$ è *negl* per dimostrare che $|Pr[E_i]|$ e $|Pr[E_{i+1}]|$ sono negligibly close;

- **Bridging steps**: questo tipo di transizioni si ha fra due giochi in cui vengono effettuate le stesse operazioni in modalità leggermente differenti, al solo fine di rendere concettualmente più comprensibili al lettore della dimostrazione gli eventi inerenti: nella pratica E_i ed E_{i+1} sono identici.

3.2.3 Un Esempio: Analisi di Sicurezza con ElGamal

Come già accennato, ElGamal è CPA-sicuro sotto l'assunzione Decisional Diffie-Hellman, secondo la quale è difficile, in senso computazionale per un avversario PPT, distinguere tra le triple (g^x, g^y, g^{xy}) e (g^x, g^y, g^z) dove x, y e z sono valori random di $\mathbb{Z}_q$. In altri termini, definendo il *DDH-advantage* ϵ_{DDH} di un avversario PPT come:

$$|Pr[x, y \xleftarrow{\$} \mathbb{Z}_q : D(g^x, g^y, g^{xy}) = 1]| - |Pr[x, y, z \xleftarrow{\$} \mathbb{Z}_q : D(g^x, g^y, g^z) = 1]|$$

l'assunzione DDH assicura che esso è negligible: su questa osservazione definiamo ora la sicurezza semantica in termini di una sequenza di giochi.

Game 0: definiamo algoritmicamente il game attack di base considerando un avversario PPT $\mathcal{A}$:

1: $x \xleftarrow{\$} \mathbb{Z}_q$; $\alpha \leftarrow g^x$;
2: $r \xleftarrow{\$} R$; $(m_0, m_1) \leftarrow \mathcal{A}(r, \alpha)$;
3: $b \xleftarrow{\$} \{0,1\}$; $y \xleftarrow{\$} \mathbb{Z}_q$;
4: $\beta \longleftarrow g^y$; $\delta \longleftarrow \alpha^y$;
5: $\varsigma \longleftarrow \delta * m_b$;
6: $\hat{b} \longleftarrow \mathcal{A}(r, \alpha, \beta, \varsigma)$;

Assumendo che $\mathcal{A}$ scelga r casualmente da un insieme R, definendo l'evento $E_0 = (b = \hat{b})$ il vantaggio ottenuto dall'avversario, detto *SS-advantage* (da semantic security), è $|Pr[E_0] - \frac{1}{2}|$.

Game 1: con una transizione basata su indistinguibilità definiamo il nuovo game come segue

1: $x \xleftarrow{\$} \mathbb{Z}_q$; $\alpha \leftarrow g^x$;
2: $r \xleftarrow{\$} R$; $(m_0, m_1) \leftarrow \mathcal{A}(r, \alpha)$;
3: $b \xleftarrow{\$} \{0,1\}$; $y \xleftarrow{\$} \mathbb{Z}_q$;
4: $\beta \longleftarrow g^y$; $z \xleftarrow{\$} \mathbb{Z}_q; \delta \longleftarrow g^z$;
5: $\varsigma \longleftarrow \delta * m_b$;
6: $\hat{b} \longleftarrow \mathcal{A}(r, \alpha, \beta, \varsigma)$;

L'unica differenza rispetto al game precedente è che calcoliamo δ con un valore z scelto uniformemente in maniera casuale da $\mathbb{Z}_q$: questo piccolo cambiamento fa sì che, dato $E_1 = (b = \hat{b})$, si abbia $Pr[S_1] = \frac{1}{2}$. Intuitivamente, m_b viene crittato non utilizzando la chiave pubblica α che poi viene fornita all'avversario, ma con δ che in questo game è un *one-time pad*: diciamo che b è indipendente, o disassociato, dall'output $\hat{b}$ dell'avversario.

Si può inoltre dimostrare che $|Pr[E_0] - Pr[E_1]| = \epsilon_{DDH}$: basta notare che nel game 0 (α, β, δ) si può esprimere come (g^x, g^y, g^{xy}), mentre nel game 1 come (g^x, g^y, g^z). Possiamo perciò definire un algoritmo distinguitore $D(\alpha, \beta, \delta)$:

1: $r \xleftarrow{\$} R$; $(m_0, m_1) \leftarrow \mathcal{A}(r, \alpha)$;
2: $b \xleftarrow{\$} \{0,1\}$; $\varsigma \longleftarrow \delta * m_b$;
3: $\hat{b} \longleftarrow \mathcal{A}(r, \alpha, \beta, \varsigma)$;

4: **if** $b = \hat{b}$ **then**
5: *output* 1
6: **else**
7: *output* 0
8: **end if**

Se l'input a D è (g^x, g^y, g^{xy}) abbiamo che

$$Pr[x, y \xleftarrow{\$} \mathbb{Z}_q \ : \ D(g^x, g^y, g^{xy}) = 1] = Pr[E_0]$$

se invece è (g^x, g^y, g^z) allora

$$Pr[x, y, z \xleftarrow{\$} \mathbb{Z}_q \ : \ D(g^x, g^y, g^z) = 1] = Pr[E_1]$$

D può quindi interpolare fra i due game ed il vantaggio DDH equivale a $|Pr[E_0] - Pr[E_1]|$. Abbiamo così dimostrato infine che

$$|Pr[E_0] - \frac{1}{2}| = \epsilon_{DDH}$$

il quale è negligible per l'assunzione DDH: partendo dal game iniziale ci siamo perciò ricondotti all'assunzione computazionale che desideravamo.

Questo semplice esempio mostra alcuni fondamenti dell'organizzazione basata su giochi, vedremo come anche i tool software per le dimostrazioni automatizzate utilizzino le tecniche indicate da Shoup, mentre ora cercheremo di capire quali siano le caratteristiche e le proprietà che un tool di questo genere dovrebbe possedere.

3.3 Sequenze di Giochi: dalla Teoria alla Pratica

Abbiamo osservato i concetti matematici alla base delle dimostrazioni game-based; in un lavoro di Bellare e Rogaway del 2004 [21] essi vengono ripresi e sviluppati definendo i giochi specificamente come se fossero programmi in cui oggetti come plaintext e ciphertext sono rappresentati come variabili e possono esservi procedure di inizializzazione per gli assegnamenti random e di finalizzazione per restituire in ouput i risultati dell'esperimento compiuto.

Anche gli avversari sono visti come frammenti di codice ed hanno la possibilità di effettuare *query* $y \leftarrow P(...)$ agli oracoli P definiti come procedure all'interno dei game, i quali si comportano come *black box* e comunicano con l'avversario. Al termine delle queries si restituisce un output che può infine

diventare il *game output* se la procedura di finalizzazione è definita, in caso contrario viene restituito l'input iniziale del game.

Aldilà della modalità espressiva però, i concetti fondamentali rimangono gli stessi: la gestione dei bad event legata al Fundamental Lemma è implementata come il settaggio di un *flag* apposito, mentre i *game-advantage* vengono sempre espressi in termini probabilistici.

Halevi, come riportato in [22], già nel 2005 pone l'attenzione sulla necessità di dover utilizzare strumenti automatizzati per gestire la complessità di prove crittografiche di dimensioni ormai ragguardevoli. Halevi cita l'esempio di una dimostrazione di oltre 23 pagine per una modalità operativa di un cifrario a blocchi. Molti aspetti 'meccanici' delle prove potrebbero ragionevolmente essere verificati in maniera automatica da un software, che fondamentalmente dovrebbe poter gestire definizioni di sequenze di giochi e permettere la definizione di trasformazioni necessarie per supportare le prove di transizione fra un gioco e l'altro.

Secondo l'articolo, il tool dovrebbe anzitutto possedere le capacità tipiche di un compilatore ovvero:

- Leggere ed interpretare come serie di istruzioni il codice dei giochi;
- Comprendere le dipendenze fra le variabili dei giochi;
- Identificare il flusso di esecuzione dei giochi nella dimostrazione;
- Permettere l'inserimento e la rimozione di *dead code* da parte dell'utente, nonché permettere manipolazioni del codice.

Le principali trasformazioni permesse sul codice dei giochi dovrebbero essere le seguenti:

- **Inserimento di bad event flag**: tramite questi flag sarebbe possibile gestire istruzioni condizionali che gestiscono il comportamento dei game in caso di bad event. Un tool evoluto potrebbe gestire anche l'inserimento automatico di bad event flag deducendoli dalle istruzioni, per esempio ad una istruzione $y \leftarrow \frac{\alpha}{x} \ mod \ q$ verrebbe settato a *true* un bad event flag nel caso che $x = 0$;
- **Coin Fixing**: la tecnica, introdotta da Bellare e Rogaway in [21], è utilizzata per assumere gli avversari come non adattabili. In pratica si

considerano variabili che dovrebbero essere date in input alla subroutine (o all'oracolo) di un certo avversario $\mathcal{A}$ come se fossero un suo output. Per esempio, se abbiamo $x \leftarrow f(y_1, ..., y_n);\ z \leftarrow \mathcal{A}(x, y_1, ..., y_n)$ possiamo sostituirla con $(x, z) \leftarrow \mathcal{A}(y_1, ..., y_n)$, purchè x non sia usata nell'output del game stesso;

- **Manipolazioni algebriche**: il tool dovrebbe supportare tipi di dato comunemente usati nei linguaggi di uso comune quali interi o booleani, ma anche altri pertinenti all'ambito crittografico, come gruppi ciclici, plaintext, ciphertext, ed altri. Il tool deve poter gestire gli operatori matematici per questi tipi di dato e le loro eventuali proprietà di commutatività, associatività e distributività. Per esempio, dati $x, y, z, k \in Int$ il software deve sapere che l'istruzione $x \leftarrow k(y+z)$ euivale a $x \leftarrow ky + kz$;

- **Operazioni che mantengono la distribuzione di probabilità**: il tool deve supportare regole per riconoscere variabili con identica distribuzione di probabilità, come x ed y nel caso si abbiano le istruzioni $x \xleftarrow{\$} \{0,1\}^n, y \leftarrow x \oplus c$; e $y \xleftarrow{\$} \{0,1\}^n, x \leftarrow y \oplus c$;

- **Operazioni che preservano la distribuzione di probabilità, bad event a parte**: se consideriamo il frammento di codice

  1: $x \xleftarrow{\$} \{0,1\}^n$;
  2: $utilizza - x - nel - game$

  sostituito con il seguente

  1: $x \xleftarrow{\$} \{0,1\}^n$;
  2: **if** $x \neq 0^n$ **then**
  3: $\quad utilizza - x - in - un - modo$
  4: **else**
  5: $\quad utilizza - x - in - un - altro - modo$
  6: **end if**

  Notiamo che lo spazio di probabilità di x viene lievemente modificato. Il tool dovrebbe quindi tenere conto delle modifiche alla distribuzione di probabilità delle variabili e quantificarle: all'istruzione 2 nel secondo frammento di codice dell'esempio il tool dovrebbe verificare che il condizionale risulta *true* con probabilità 2^{-n} basandosi sull'entropia delle variabili, ossia sulla quantità di informazione necesaria per rap-

presentarle, in questo caso n. Ciò è particolarmente importante in quanto le istruzioni condizionali sono generalmente usate per settare bad event flag.

Una libreria di trasformazioni così definita potrebbe inoltre arricchirsi sulla base delle necessità degli utenti, aggingendo per esempio nuovi tipi di dato ed operatori o nuove trasformazioni su di esse man mano che vengono scoperte nuove dimostrazioni.

Un'altra caratteristica utile per un tool completo è la generazione automatica di template: nel nostro ambito per esempio, se si volesse dimostrare la proprietà di CPA-security di un nuovo cifrario, si potrebbe velocizzare il processo generando automaticamente il codice, ovviamente incompleto, dei giochi inerenti al *CPA-indistinguishability-experiment*.

Un altro aspetto cruciale per l'usabilità del software riguarda la progettazione dell'interfaccia utente, in particolare:

- La scrittura del codice deve essere semplificata e possibilmente ad alto livello per quanto riguarda i costrutti crittografici: se un utente definisce per esempio $var\ pk : public - key - type = \alpha \leftarrow g^x$; deve essere chiaro che si tratta di una chiave pubblica, che può essere poi passata ed utilizzata da un avversario, ecc...;

- Il codice deve essere interamente comprensibile e facilmente modificabile dall'utente;

- Il tool dovrebbe essere in grado di restituire l'output del codice di un game in un formato, ad esempio LaTeX, per poter definire eventualmente parti della prova non formalizzabili dal software;

- Data la struttura delle dimostrazioni game-based, una rappresentazione grafica delle sequenze di giochi, opportunamente caratterizzata, faciliterebbe la comprensione delle dimostrazioni all'utente.

Vedremo come EasyCrypt riesca a soddisfare alcune di queste proprietà, pur essendo ancora ad uno stadio preliminare del suo sviluppo. Prima di descriverne gli aspetti principali però, daremo alcuni cenni al sistema che sottende alle dimostrazioni di EasyCrypt, CoQ.

3.3.1 CoQ: un Proof Assistant per EasyCrypt

Coq [23] è un tool per computer, attualmente alla versione 8.4, impiegato per la verifica di dimostrazioni di teoremi e per la verifica formale di programmi. CoQ è l'acronimo di Calculus of Constructions, un verifier ideato da Thierry Coquand (nomen omen) nel 1984 per l'analisi dei tipi sul λ-calcolo tipizzato semplice, il sistema si è poi evoluto fino a diventare il *theorem prover* interattivo che è oggi.

Si tratta quindi di un prodotto maturo ed ampiamente utilizzato per lo sviluppo di programmi che necessitano di una rigorosa verifica di proprietà, come nell'ambito delle telecomunicazioni, dei prodotti finanziari o, nel nostro caso, dei protocolli crittografici.

Il sistema è implementato con il linguaggio funzionale OCAML e di recente è stato anche sviluppato un ambiente di sviluppo, CoQIDE, per semplificare l'interazione con gli utenti.

CoQ possiede un linguaggio, *Gallina*, per la definizione dei programmi con termini, tipi e dimostrazioni, mentre utilizza un linguaggio comandi differente, denominato *Vernacular*, per la comunicazione fra l'utente ed il sistema: ad esempio possiamo caricare moduli con *Require Import Bool* o controllare la corretta tipizzazione di un termine t con il comando *Check t*.

CoQ come accennato consente di definire λ-termini e tipi nonché di effettuare computazioni mediante diverse regole di riduzione, come la β-riduzione o la δ-riduzione. Una caratteristica importante di CoQ è che si tratta di un formalismo fortemente normalizzante: ciò assicura che le computazioni abbiano sempre termine, a scapito di una piccola parte di potere espressivo, essendovi infatti funzioni descrivibili in CoQ ma non riducibili.

Tra i tipi semplici a disposizione abbiamo i più comuni tipi induttivi, quali numeri naturali e booleani, nonché tipi più complessi, come i tipi freccia $A \to B$ dove A, B sono a loro volta tipi o come il tipo polimorfico *list* A, lista omogenea di elementi di tipo A. Il sistema di tipi, come vedremo, è ereditato da Easycrypt, il quale beneficia della possibilità di avere espressioni e comandi ben tipati.

Il Calculus of Inductive Constructions possiede una potenza espressiva ben superiore a quella del λ-calcolo tipizzato semplice: è infatti possibile anche esprimere proposizioni di tipo *Prop* che rappresentano formule, ad esempio $a : A$, se A è una proposizione, viene interpretato da CoQ come 'a è una prova di A', mentre a è un termine di prova. Le proposizioni possono

essere dimostrate solo utilizzando assunzioni di altre proposizioni, che possono essere definite come ipotesi, assiomi o teoremi, definibili con sintassi differenti.

I termini di prova possono diventare molto complessi: per questo si considera una proposizione A da dimostrare come un *goal* che può essere ridotto in molteplici goal più semplici o, se possibile, direttamente risolto mediante comandi definiti *tactics*: come vedremo, anche EasyCrypt utilizza questa nozione per gestire le prove e la corretta combinazione delle tactics a disposizione risulta essere fondamentale per per poter completare le dimostrazioni.

CoQ è alla base di EasyCrypt per quanto riguarda la formalizzazione della teoria matematica, probabilistica e computazionale dei protocolli crittografici dei quali EasyCrypt può dimostrare proprietà di sicurezza. Anche il linguaggio di definizione dei giochi di EasyCrypt che vedremo più specificamente nelle prossime sezioni, pWhile, è costruito su CoQ.

3.4 EasyCrypt: un Tool Pratico

Nella precedente sezione abbiamo potuto constatare l'esistenza di strumenti per prove formali e rigorose dal punto di vista matematico, tuttavia le osservazioni sottolineate in precedenza non possono essere risolte con l'aiuto di CoQ, almeno per quanto riguarda l'ambito crittografico.

Anzitutto, CoQ è un sistema di grande potenza espressiva, ma complesso da apprendere e soprattutto *general purpose*: la verifica di dimostrazioni in specifici ambiti presuppone la definizione formale di tutta la teoria che sta alla base delle dimostrazioni, portando così ad avere prove molto verbose e scarsamente comprensibili da parte di chi deve utilizzare e sviluppare protocolli crittografici.

Per questo motivo Barthe, Zanella ed altri [24] [25], nel 2009 hanno presentato un *framework* denominato *CertiCrypt* costruito su CoQ, in grado di mettere a disposizione una serie di principi di ragionamento e tecniche utilizzabili specificamente dai crittografi alo scopo di verificare prove game-based. CertiCrypt, che successivamente verrà sviluppato e diventerà EasyCrypt, mette a disposizione le seguenti *features*:

- **pWhile**: si tratta di un linguaggio di programmazione imperativo con estensioni probabilistiche mediante il quale è possibile definire programmi che rappresentano dimostrazioni composte da giochi. L'utente

ha la possibilità di usare assegnamenti probabilistici, strutture dati e tipi di dato specifici e procedure come in ogni linguaggio imperativo general purpose, ma è possibile anche definire assiomi, operatori e tipi di dato appositi per dimostrazioni crittografiche. Data la possibilità di definire anche avversari ed oracoli, è anche specificato un modello dell'avversario per definire formalmente avversari PPT;

- **pRHL**: il supporto alla *probabilistic Relational Hoare Logic* è necessario per dimostrare le transizioni fra giochi, esprimendo specifici giudizi riguardanti coppie di giochi. CertiCrypt mette a disposizione un *tactic language* apposito per verificare la validità dei giudizi: grazie ad esso si possono ottenere una serie di *verification conditions* espresse in un linguaggio del primo ordine, senza riferimenti probabilistici, anche dette SMT (*Satisfiability Modulo Theories*) *instances*;

- Uno degli obiettivi definiti dal progetto di CertiCrypt originale era quello di generare, oltre alle verification conditions, anche dei file CoQ verificabili indipendentemente con il framework CertiCrypt ma, al momento, lo sviluppo di tale meccanismo è ancora in fase preliminare e non disponibile in EasyCrypt.

EasyCrypt, inizialmente concepito solo come *front-end* per CertyCrypt, ad oggi ne eredita tutte le caratteristiche ed è ancora più accessibile alla comunità di crittografi grazie ad un linguaggio di input più conciso ed una maggiore capacità di automazione fornita dal supporto a solver SMT. Una volta ottenute una serie di verification conditions per provare le transizioni fra giochi, EasyCrypt permette infatti di utilizzare automaticamente alcuni solver SMT esterni quali Alt-Ergo, Cvc3, Simplify ed altri, anzichè utilizzare CoQ. Essi sono in grado di risolvere automaticamente le verification conditions e completare così le dimostrazioni.

Vedremo come EasyCrypt sia stato utilizzato per effettuare numerose dimostrazioni riferibili a sistemi crittografici differenti di cui vedremo alcuni esempi. Nel resto della sezione provvederemo a definire più precisamente il funzionamento dei meccanismi sopracitati nonché a descrivere un esempio di prova concreto.

3.4.1 Il Linguaggio pWhile

Il linguaggio pWhile, utilizzato in EasyCrypt per la definizione di giochi come programmi, deriva dal framework CertyCrypt ed è l'estensione probabilistica di un linguaggio imperativo. La sintassi è rappresentata in Fig. 3.1:

$$
\begin{array}{rll}
\mathcal{C} ::= & \mathsf{skip} & \text{nop} \\
| & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\
| & \mathcal{V} \xleftarrow{\$} \mathcal{DE} & \text{probabilistic assignment} \\
| & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
| & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{loop} \\
| & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
| & \mathcal{C};\ \mathcal{C} & \text{sequence}
\end{array}
$$

Fig. 3.1: Sintassi di pWhile

dove $\mathcal{V}$ è un set di identificatori di variabili, $\mathcal{P}$ un set di identificatori di procedure, $\mathcal{E}$ un set di espressioni deterministiche e $\mathcal{DE}$ un insieme di espressioni probabilistiche: con questi elementi è possibile definire istruzioni $\mathcal{I}$ e comandi $\mathcal{C}$ dei giochi.

Le espressioni probabilistiche vengono valutate come distribuzioni uniforme da cui vengono campionati casualmente dei valori, ad esempio:

```
var b : bool;
b={0,1};
```

qui definiamo legalmente una variabile booleana b e le assegniamo un valore casuale tra 0 ed 1.

I tipi base per cui EasyCrypt offre supporto nativo sono *unit*, *bool*, *int*, *real*, *bitstring*, i tipi polimorfici *list* e *map* ed infine i tipi *sum* ed *option*. Si possono definire costanti valide globalmente nel programma oltre che procedure da utilizzare all'interno dei giochi. I giochi possono essere definiti *ex novo* oppure è possibile ridefinire una o più delle loro procedure.

Gli avversari vengono definiti esternamente ai giochi e si comportano come procedure: è possibile specificare gli elementi tipati in input ed il tipo dell'output che restituiscono; inoltre si può anche dichiarare il tipo dell'oracolo che utilizza. Vediamo nell'esempio seguente:

```
adversary A1_sign(pk:pkey) : plaintext * plaintext {group ->
    message}.
```

Si dichiara un avversario che riceve in input una chiave pubblica e restituisce una coppia di plaintext, inoltre si specifica che esso utilizza un oracolo di tipo $group \rightarrow message$: in un game potremo avere quindi

```
abs A1 = A1_sign{oracle_func}
```

dove *oracle_func* è una funzione di tipo $group \rightarrow message$ che funge da oracolo. Proprio come le procedure, anche gli avversari possono essere dichiarati come astratti all'interno dei game.

Si può notare come, nella dichiarazione di A1, l'elemento in input sia di tipo *pkey*, che non è citato fra i tipi nativi: si tratta di un tipo astratto, che è possibile definire ed istanziare nello *scope* globale del programma. Vi è la possibilità di istanziare un tipo astratto direttamente con un tipo base con altri tipi astratti definiti dagli utenti. Oltre ai tipi astratti è possibile anche dichiarare degli operatori per manipolarli, la cui implementazione è affidata a CoQ; per descrivere le proprietà di tipi astratti ed operatori si possono definire assiomi e lemmi basati su formule del primo ordine. Vediamo per esempio un frammento di codice per la costruzione del tipo chiave pubblica *pkey* nel caso di una dimostrazione con il cifrario ElGamal:

```
type group.

type pkey = group.

op [*] : (group, group) -> group as group_mult.
op [^] : (group, int) -> group   as group_pow.

axiom group_pow_add :
 forall (x, y:int), g ^ (x + y) = g ^ x * g ^ y.

axiom group_pow_mult :
 forall (x, y:int),  (g ^ x) ^ y = g ^ (x * y).
```

Il tipo *pkey* è istanziato con un altro tipo astratto, il tipo *group*, riferito al gruppo ciclico moltiplicativo $\mathbb{Z}_q$ citato nel Capitolo2 per ElGamal CPA-sicuro. Vengono definiti gli operatori di moltiplicazione fra elementi del gruppo e di potenza sulla base degli operatori *group_mult* e *group_pow* im-

plementati in CoQ e si stabiliscono delle proprietà tramite assiomi che vengono verificati da EasyCrypt utilizzando uno degli SMT solver disponibili.

Per una descrizione più precisa ed approfondita della sintassi dei costrutti qui descritti rimandiamo al manuale di EasyCrypt [26].

Un programma pWhile c viene interpretato da EasyCrypt come una funzione $[\![c]\!]$ che mappa da una memoria iniziale ad una sottodistribuzione sulle memorie finali che risultano eseguendo c; essendo pWhile fortemente tipizzato, il mapping sarà effettuato correttamente solo tra variabili e valori del tipo appropriato, senza opportunità di applicare *cast* o tecniche similari. Considerando un insieme di memorie finali finito, un mapping assegna ad ogni memoria una probabilità nell'intervallo $[0, 1]$ tale che il totale delle memorie ha come *upper bound* 1: la sottodistribuzione delle memorie viene rapresentata da una struttura apposita, la monade $\mathcal{D}$ di Audebaud e Paulin formalizzata in CoQ. La funzione è perciò definita come

$$[\![c]\!] \ : \ m \to \mathcal{D}(m)$$

dove m rappresenta la memoria iniziale. Per le regole di semantica denotazionale di $[\![c]\!]\ m$ definite mediante gli operatori di $\mathcal{D}$ rimandiamo alla Sezione 2.2 di Zanella [24].

In generale, dati un programma c rappresentante un game, una memoria iniziale m ed un evento E riguardante l'output del game definiamo $Pr[c, m \ : \ E]$ come la probabilità di E in riferimento alla sottodistribuzione indotta da $[\![c]\!]\ m$.

3.4.2 Transizioni fra Giochi con pRHL

In EasyCrypt i meccanismi di reasoning per le transizioni fra giochi sono affidati a pRHL, un'estensione probabilistica della logica di Hoare relazionale definita da Benton nel 2004. In questa logica abbiamo le seguenti definizioni:

> Un giudizio è espresso nella forma
>
> $$\models c_1 \sim c_2 \ : \ \Psi \Rightarrow \Phi$$
>
> dove c_1 e c_2 sono programmi probabilistici e Ψ e Φ sono rispettivamente *pre-* e *post*-condition intese come formule relazionali.

Le formule relazionali Ψ e Φ sono esprimibili nella grammatica seguente:

$$\Psi, \Phi ::= \ e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \Longrightarrow \Phi \mid \forall x.\Phi \mid \exists x.\Phi$$

in cui e è un'espressione booleana contenente o variabili logiche opportunamente legate o variabili di programma taggate con $\langle 1\rangle$ o $\langle 2\rangle$ per indicare a quale programma appartengono. La keyword res è utilizzata per indicare il valore di ritorno di una procedura e può essere utilizzata nelle formule relazionali come una variabile di programma.

Le formule relazionali sono interpretate da EasyCrypt come relazioni fra memorie m_1 ed m_2: per esempio la formula $\forall y.\ y + x\langle 1\rangle - 1 \leq k\langle 2\rangle$ è interpretata come la relazione

$$\Phi = \{(m_1, m_2) \mid \forall y.\ y + \ m_1(x) - 1 \leq m_2(k)\}$$

Definite pre- e post- condition possiamo dare alcune ulteriori definizioni.

Un giudizio è ritenuto valido sse

$$\forall m_1, m_2.\ m_1\ \Psi\ m_2 \Longrightarrow ([\![c_1]\!]\ m_1)\Phi([\![c_2]\!]\ m_2)$$

ovvero per ogni coppia di memorie m_1, m_2 che soddisfa la pre-condition Ψ, le sottodistribuzioni $([\![c_1]\!]\ m_1)$ e $([\![c_2]\!]\ m_2)$ soddisfano Φ.

Se consideriamo che c_1 e c_2 sono programmi che rappresentano procedure di due game differenti, vale la seguente proprietà:

Se

$$\models c_1 \sim c_2\ :\ \Psi \Rightarrow \Phi$$

allora

$$\forall m_1, m_2, A, B.\ m_1\ \Psi\ m_2, \Phi \rightarrow (A\langle 1\rangle \leftrightarrow B\langle 2\rangle)$$

$$\Longrightarrow Pr[c_1, m_1\ :\ A] = Pr[c_2, m_2\ :\ B]$$

ossia per ogni coppia di memorie m_1, m_2 di due giochi differenti e per ogni coppia di eventi A, B tali che se vale Φ tutte le memorie m_1', m_2' che soddisfano A soddisfano anche B e viceversa, abbiamo la stessa probabilità dei due eventi in riferimento alle sottodistribuzioni indotte.

Analogamente per le disuguaglianze, con pRHL vale

Se

$$\models c_1 \sim c_2 \ : \ \Psi \Rightarrow \Phi$$

allora

$$\forall m_1, m_2, A, B.\ m_1\ \Psi\ m_2, \Phi \rightarrow (A\langle 1\rangle \rightarrow B\langle 2\rangle)$$

$$\Longrightarrow Pr[c_1, m_1 \ : \ A] \leq Pr[c_2, m_2 \ : \ B]$$

dove $\Phi \rightarrow (A\langle 1\rangle \rightarrow B\langle 2\rangle)$ indica che tutte le memorie m_1', m_2' che soddisfano A soddisfano anche B.

Grazie alle ultime due proprietà enunciate è possibile ragionare probabilisticamente sugli eventi riguardanti i giochi partendo dalla validità dei giudizi. EasyCrypt implementa anche un banale meccanismo per calcolare il limite di probabilità: in generale riesce a calcolare come un valore, scelto uniformemente in maniera casuale da un insieme $\mathcal{T}$, è uguale ad una espressione arbitraria con probabilità $\frac{1}{|\mathcal{T}|}$ mentre, se si tratta di una lista con n valori, è al massimo $\frac{n}{|\mathcal{T}|}$.

Abbiamo visto nella sezione precedente il Fundamental Lemma, per il quale dati due eventi A in G_1 e B in G_2 è possibile trovare un *upper bound* a $|Pr[A] - Pr[B]|$ se G_1 e G_2 si comportano in maniera identica fino a che non accade un bad event F. EasyCrypt, seguendo anche le linee guida espresse da Halevi ed esaminate in precedenza, consente di ragionare sulle probabilità di G_1 e G_2 grazie al Fundamental Lemma. Nella pratica, è possibile settare nei programmi un flag *bad* a true: EasyCrypt, verificando che il codice di G_1 e G_2 differisce soltanto a partire dalle istruzioni successive al settaggio del flag *bad*, può applicare il Fundamental Lemma. Dato che $|Pr[A] - Pr[B]|$ è limitato dalla probabilità del bad event F, EasyCrypt applica regole apposite sul codice per calcolare un upper bound di $Pr[F]$ induttivamente.

Generalmente il flag *bad* viene settato a *true* soltanto nelle procedure che vengono passate come oracoli agli avversari, nel caso in cui venga superato il limite massimo di *queries* che un avversario può inviare all'oracolo, se è conosciuto; anche in questo caso EasyCrypt può determinare in alcuni casi l'upper bound di $Pr[F]$ per induzione strutturale.

Finora abbiamo definito in via teorica i giudizi pRHL e stabilito quando possono essere considerati validi, ma non abbiamo descritto alcun meccanismo pratico impiegato da EasyCrypt per accertarne la validità: abbiamo

però accennato in precedenza al fatto di avere a disposizione un tactic language che consente di ridurre i giudizi ad una serie di verification conditions gestibili poi automaticamente grazie ai solver SMT. Esamineremo al meglio questo linguaggio e le tattiche a disposizione nel Capitolo 4, in cui analizzeremo a livello pratico come provare le transizioni fra giochi.

3.4.3 ElGamal: un Esempio di Dimostrazione con Easycrypt

Abbiamo già descritto le caratteristiche del cifrario a chiave pubblica ElGamal ed abbiamo anche visto una dimostrazione concettuale game-based della sicurezza CPA elaborata da Shoup; ne vedremo ora un'implementazione in EasyCrypt in modo da poter illustrare l'utilizzo pratico delle features del tool presentate nell'ultima sezione e poter comparare l'implementazione con la prova su carta.

Il game di partenza è quello che rappresenta il CPA indistinguishability experiment, di cui omettiamo parti di codice non utili alla comprensione, come le dichiarazioni di variabili:

```
Game INDCPA:
 (pk,sk) = KG();
 (m0,m1,sigma) = A1(pk);
 b = {0,1};
 c = Enc(pk, b ? m0 : m1);
 b' = A2(c, sigma);
 return (b = b');
```

Lo *challenger* genera una chiave segreta ed una pubblica e passa quest'ultima all'avversario $A1$, il quale restituisce due plaintext m_0, m_1; lo challenger sceglie casualmente un valore booleano 0 o 1 e critta il plaintext corrispondente, il quale viene poi passato all'avversario $A2$. Si può notare che l'avversario CPA sia modellato mediante $A1$ ed $A2$: entrambe condividono lo stesso stato, utilizzando *sigma* come variabile di stato. Definiamo l'IND-CPA-*advantage* dell'avversario come $|Pr[INDCPA : res] - \frac{1}{2}|$, concettualmente identico all'SS-advantage definito con il $Game_0$ di Shoup in precedenza.

Abbiamo poi due game DDH0 e DDH1:

```
Game DDH0:
 x = [0..q-1];
 y = [0..q-1];
 d = B(g^x, g^y, g^(x*y));
 return d;
```

```
Game DDH1:
 x = [0..q-1];
 y = [0..q-1];
 z = [0..q-1];
 d = B(g^x, g^y, g^z);
 return d;
```

che esplicitano il problema DDH; esprimiamo il DDH-advantage come

$$Pr[DDH_0 : res] - Pr[DDH_1 : res]$$

La procedura B chiamata dai giochi DDH corrisponde al distinguitore D di Shoup ed è così definita:

```
fun B(alpha,beta,gamma:group) : bool = {
    (*Dichiarazione variabili*)
    (m0, m1, sigma) = A1(alpha);
    b = {0,1};
    c = (beta, gamma * (b ? m0 : m1));
    b' = A2(c, sigma);
    return (b = b');
  }
```

Abbiamo infine un game IND:

```
Game IND:
 x = [0..(q - 1)];
 y = [0..(q - 1)];
 (m0, m1, sigma) = A1(g^x);
 z = [0..(q - 1)];
 b' = A2((g^y, g^z), sigma);
 b = {0,1};
 return b = b';
```

simile a INDCPA ma in cui è esplicita la disassociazione tra b scelto dallo challenger e l'output b' dell'avversario, essendo b scelto dopo l'esecuzione di $A2$.

L'intera struttura della dimostrazione per riduzione è rappresentata in Fig. 3.2.

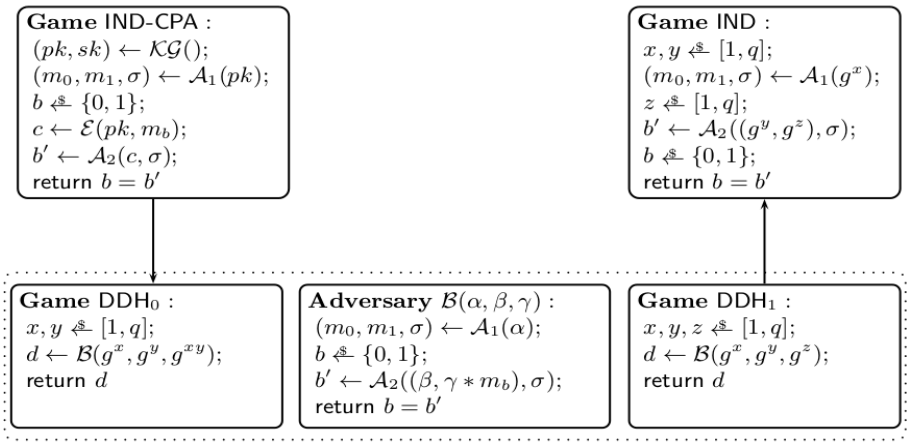

Fig. 3.2: Prova di sicurezza CPA di ElGamal con sequenza di giochi

Come si può evincere dall'illustrazione, occorre sostanzialmente verificare due transizioni fra giochi, ovvero la validità di due giudizi pRHL. Il primo giudizio stabilisce che la distribuzione di probabilità dell'output del game INDCPA (data da $b = b'$) è uguale a quella di d restituito da DDH0:

$$\models INDCPA \sim DDH_0 : true \Rightarrow res\langle 1\rangle = res\langle 2\rangle$$

con cui possiamo stabilire la riduzione di INDCPA a DDH0:

$$Pr[INDCPA : res] = Pr[DDH_0 : res]$$

Il secondo giudizio è invece riferito alla transizione dal game IND a DDH_1:

$$\models DDH_1 \sim G : true \Rightarrow res\langle 1\rangle = res\langle 2\rangle$$

Nonostante la prova appaia intuitiva, vedremo nel Capitolo 4 come EasyCrypt non riesca a provare automaticamente giudizi come questo ed occorra sfruttare opportunamente una serie di tattiche. Una volta fatto ciò verifichiamo quindi che

$$Pr[DDH_1 : res] = Pr[IND : res]$$

Possiamo inoltre calcolare il valore di $Pr[IND : res]$ sfruttando i meccanismi di computazione basilari di probabilità di EasyCrypt:

```
claim Independent : IND.Main[res] = 1%r / 2%r by compute
```

dove Main è intesa come la procedura che fa da corpo al game G. In questo caso, la computazione riesce in quanto b e b' sono totalmente disassociati ed EasyCrypt può verificarlo a livello di codice. A questo punto, unificando i giudizi verificati otteniamo che

$$|INDCPA.Main[res] - \frac{1}{2}| = |DDH0.Main[res] - DDH1.Main[res]|$$

che equivale ad affermare che INDCPA-advantage=DDH-advantage.

Nell'esempio abbiamo potuto constatare come la definizione dei giochi e delle transizioni fra di essi sia effettivamente intuitiva e vicina a quella espressa concettualmente da Shoup. Questo è possibile in primo luogo grazie alla natura *specific-domain-purpose* di EasyCrypt, grazie alla quale anche senza possedere una vasta formazione in ambito di prove formali *computation-aided* è possibile usare costrutti predefiniti come avversari, tipi chiave, ciphertext ed altri senza doversi occupare della loro implementazione. Dando appunto per scontate queste nozioni, il prossimo capitolo sarà dedicato all'utilizzo pratico di EasyCrypt, anche in relazione all'applicazione del tool con il caso di studio delle mixnets.

Capitolo 4

EasyCrypt: Prove Pratiche e Mixnets

4.1 Bridging Steps in EasyCrypt

Nel precedente capitolo, abbiamo descritto EasyCrypt ed abbiamo definito formalmente come esprimere le transizioni fra giochi mediante giudizi pRHL. Abbiamo anche visto come in EasyCrypt esistano meccanismi di base per computare direttamente le probabilità di alcuni eventi legati a giochi, come nel caso del game G nell'esempio di ElGamal. Abbiamo inoltre a disposizione i già citati meccanismi legati ai bad event F, in particolare per quanto riguarda le verifiche sul codice operate da EasyCrypt per calcolare $Pr[F]$ per induzione strutturale sul codice.

Tuttavia non abbiamo descritto le modalità pratiche con cui EasyCrypt gestisce la maggior parte delle transizioni generalmente usate nelle dimostrazioni game-based, ovvero i *bridging steps*.

I bridging steps sono spesso semplici da dimostrare concettualmente ma, in particolare per quanto concerne protocolli pratici, risultano piuttosto complessi da formalizzare in maniera precisa: occorre infatti stabilire se le distribuzioni in ouput di due programmi differenti siano identiche. Oltretutto, occorre tenere presente che piccole discrepanze a livello concettuale possono comportare imponenti differenze a livello di implementazione nel codice.

Nella definizione ideale di Halevi [22] tali procedimenti di dimostrazione dovrebbero automatizzati, date le opportune assunzioni ed una corretta

definizione dei giochi in forma di programmi, ma si tratta in realtà di un requisito estremamente complesso da soddisfare, che si scontra con i limiti attuali dell'analisi statica dei programmi.

EasyCrypt riesce solo parzialmente nell'obiettivo, mettendo a disposizione un potente *tactic language* per la riduzione dei giudizi in pRHL a verification conditions risolvibili automaticamente dai solver SMT. Nella prima parte del capitolo analizzeremo il tactic language ed illustreremo gli sforzi fatti per implementarvi tattiche automatizzate e semplici da utilizzare per gli utenti.

4.2 Verifica dei Giudizi con Tattiche

In EasyCrypt possiamo esprimere giudizi tramite la sintassi:

```
equiv Fact : G1.f1 ~ G2.f2 : Pre ==> Post.
```

dove $f1$ ed $f2$ sono due procedure dei rispettivi game. Riutilizzando l'esempio di ElGamal del Capitolo 3 possiamo vedere come il giudizio

```
equiv CPA_DDH0 : INDCPA.Main ~ DDH0.Main : true ==> ={res}.
```

in cui $= \{res\}$ sta per $res\{1\} = res\{2\}$ venga automaticamente trasformato da EasyCrypt nel seguente goal:

```
pre = true
stmt1 =
1 : (sk, pk) = KG ();
2 : (m0, m1) = A1 (pk);
3 : b = {0,1};
4 : mb = if b then m0 else m1;
5 : c = Enc (pk, mb);
6 : b' = A2 (pk, c);
stmt2 =
1 : x = [0..q - 1];
2 : y = [0..q - 1];
3 : d = B (g ^ x, g ^ y, g ^ (x * y));
post = (b{1} = b'{1}) = d{2}
```

dove $stmt1$ e $stmt2$ sono i *bodies* delle procedure Main dei giochi INDCPA e DDH0.

I goal sono visualizzabili su terminale al momento dell'interpretazione dei file '*.ec' che rappresentano una prova completa comprendente le defini-

zioni di operatori, tipi, assiomi ed avversari utilizzati, la sequenza di giochi composti da una procedura Main più altre eventuali procedure ed infine una serie di equivalenze/giudizi con relativi statement conclusivi.

Una volta definito un giudizio come sopra, l'interprete di EasyCrypt si attende in input una serie di tattiche per guidare la dimostrazione. L'applicazione di ogni tattica ad un goal può generare sia una serie di verification conditions sotto forma di formule del primo ordine che vengono passate a solver SMT in grado di risolverle automaticamente, che una serie di ulteriori *subgoal* che devono essere gestiti dall'utente mediante altre tattiche. Avremo quindi una dimostrazione nella forma:

```
equiv CPA_DDH0 : INDCPA.Main ~ DDH0.Main : true ==> ={res}.
proof.
(*
 Serie di tattiche
 *)
save.
```

In una prova corretta devono essere utilizzate le opportune tattiche in modo da non lasciare alcun *pending subgoal* ed ottenere solo verification conditions per il solver SMT.

Esistono tre diversi tipi di tattiche in EasyCrypt:

- **Basic tactics**: ognuna di esse implementa una specifica regola pRHL derivata da CertiCrypt;
- **Program Transformation tactics**: permettono di gestire trasformazioni del codice come *inline* di procedure o *swap* di istruzioni;
- **Automated tactics**: al fine di facilitare le prove all'utente si hanno a disposizione tattiche semi-automatizzate che combinano propriamente diverse tattiche di base;

Vedremo ora una panoramica più dettagliata di queste tre tipologie di tattiche.

4.2.1 Basic Tactics

Abbiamo già accennato nel Capitolo 3 al concetto di validità di un giudizio in relazione a due programmi pWhile c_1 e c_2; sono definite, proprio per

induzione strutturale su c_1 e c_2, diverse regole pRHL come ad esempio quella basilare di composizione sequenziale:

$$\frac{\vdash c_1 \sim c_2 \ : \ \Psi \Rightarrow \Theta \quad \vdash c_1' \sim c_2' \ : \ \Theta \Rightarrow \Phi}{\vdash c_1; c_1' \sim c_2 : c_2' \ : \ \Psi \Rightarrow \Theta}$$

Alcune regole risultano essere complesse ed espresse in una logica di ordine superiore, mentre altre anziché essere simmetriche come quella dell'esempio hanno anche una versione *one-sided*, left o right a seconda se si riferiscano rispettivamente a c_1 o a c_2, come la regola $[Cond]$ one-sided left:

$$\frac{\vdash c_1; c \sim c' \ : \ \Psi \wedge e\langle 1 \rangle \Rightarrow \Phi \quad \vdash c_2; c \sim c' \ : \ \Psi \wedge \neg e\langle 1 \rangle \Rightarrow \Phi}{\vdash if \ e \ then \ c_1 \ else \ c_2 \sim c' \ : \ \Psi \Rightarrow \Theta}$$

A queste regole pRHL corrispondono tattiche precise che descriveremo brevemente, soffermandoci sugli effetti pratici che esse sono in grado di applicare: per una definizione formale delle regole pRHL associate alle tattiche ed una descrizione comprensiva di ulteriori esempi rimandiamo al manuale di EasyCrypt [26] alla Sezione 2.4.

Descriviamo di seguito le principali tattiche di base comprensive di sintassi: ci riferiremo al programma *left-hand* del giudizio come LH ed a quello *right-hand* come RH per brevità.

Tattica App - sintassi: $app \ m \ n \ p$
Applica la regola di composizione sequenziale (vista nel primo esempio) applicata alle prime m istruzioni di LH ed alle prime n di RH, con p come Θ. Vengono generati due subgoal, per verificare rispettivamente $\vdash c_1 \sim c_2 \ : \ \Psi \Rightarrow \Theta$ e $\vdash c_1 \sim c_2 \ : \ \Theta \Rightarrow \Phi$.
Esempio: usando $app \ 1 \ 1 \ = \{pk, sk\}$ sul goal

```
pre = true
stmt1 =
1 : (sk, pk) = KG ();
2 : (m0, m1) = A1 (pk);
3 : b = {0,1};
      ...
stmt2 =
1 : (sk, pk) = KG ();
2 : (m0, m1) = A1 (pk);
```

```
3 : b = {0,1};
      ...
post = (b{1} = b'{1}) = d{2}
```

otteniamo un subgoal per verificare le istruzioni 1 dei due statement con post-condition = $\{pk, sk\}$ ed uno per la verifica delle restanti istruzioni con la post-condition originale e = $\{pk, sk\}$ come pre-condition.

Tattica Rnd - sintassi: *rnd* [*side*] [*dir*] [$(fct)|(fct)(fct)$]
Questa tattica è usata per verificare programmi con istruzioni di assegnamento probabilistico ed implementa regole pRHL in versione one-sided o two-sided. I parametri opzionali sono *side*, *dir* per indicare se l'istruzione random si trovi al top al bottom delle istruzioni nel goal attuale e per la versione two-sided, un'involuzione f o una biiezione (f, g).

Nella pratica la versione one-sided genera un nuovo goal senza l'assegnamento random ma con una post-condition opportuna: se ad esempio abbiamo un'istruzione in LH $y = [0..q-1]$ e nella post-condition è presente $y\{1\}$ allora sarà aggiunta la formula $forall(y : int), 0 <= y => y <= q-1$.

La versione two-sided è più complessa: sovente viene utilizzata a seguito della tattica *App* per dimostrare il subgoal Θ intermedio in presenza di assegnamenti probabilistici in LH ed RH: in sostanza l'involuzione o la coppia di funzioni fra loro inverse, nella forma $var \rightarrow exp$, devono fornire una giustificazione valida per la post-condition del goal corrente. Per esempio se abbiamo il goal, dove %% è l'operatore di modulo:

```
pre   = true
stmt1 =   1 : x = [0..q - 1];
stmt2 =   1 : x = [0..q - 1];
          2 : y = [0..q - 1];
post  = x{1} = (x{2} + y{2}) %% q
```

per giustificare la post-condition occorrerà definire una coppia di funzioni che permettano di considerare lo *stmt*2 come se vi fosse soltanto un assegnamento probabilistico perciò utilizziamo la tattica:

```
rnd (w - (w - x{2}) %% q),(w -> (w + x{2}) %% q).
```

generando una verbosa post-condition in cui si esprimono le verification conditions per controllare che le due funzioni f, g date formino una biiezione e per provare che f applichi una giustificazione corretta.

Tattica Case - sintassi: $case\ [side]\ :\ prog - expr$
Genera due subgoal a partire da quello corrente: nel primo si ha come pre-condition l'intersezione logica fra la pre-condition originale e $prog - expr$, mentre nel secondo fra pre-condition originale e $\neg prog - expr$, suddividendo la prova in due rami alternativi. Nella versione two-sided occorre che $prog - expr$ sia valida sia in LH che RH.

Tattica If - sintassi: $if\ [side]$
Si tratta dell'implementazione della regola pRHL $[Cond]$ citata in versione one-sided: a partire dal goal corrente se ne generano due per verificare, tramite le opportune pre-condition, la validità della condizione nell'if per il primo subgoal e la validità della condizione negata per il secondo subgoal. Nel primo subgoal generato avremo negli *stmt* il primo ramo dell'istruzione condizionale, nel secondo subgoal il secondo ramo.

Tattica While - sintassi: $while\ [side]\ [dir]\ :\ Inv\ [:\ variant, bound]$
La tattica while viene utilizzata quando nel goal corrente occorre verificare un loop *while*, one-sided o two-sided. Nel caso della versione two-sided deve essere passata soltanto una formula relazionale Inv utilizzata come invariante del ciclo: vengono generati due subgoal, uno con le opportune pre- o post-condition (a seconda del valore di *dir*) per la verifica dell'invariante e del codice all'interno dei loop enunciato negli *stmt* ed un altro che, controllando sempre la validità di Inv, verifica il resto dei programmi esterni al loop.

La versione one-sided verifica solo un loop, in LH o RH: devono essere fornite in aggiunta, per controllare la terminazione del ciclo, una formula relazionale *variant* intesa come variante decrescente all'interno del loop ed un *bound* inteso come valore di limite inferiore. Come per la versione two-sided sono generati due subgoal, uno per il corpo del ciclo ed uno per il resto del programma.

Tattica Call - sintassi: $call\ [usingId]$
Una tattica dotata di un buon grado di automatizzazione è *call*: essa è utilizzabile solo in versione two-sided ed esegue le chiamate a procedura in LH ed RH, al bottom del goal corrente, ed è in grado di utilizzare un giudizio già verificato in precedenza, denominato con Id e riguardante proprio le due

procedure nel goal corrente. In concreto, *call* consente di utilizzare prove di giudizi già completate in nuove dimostrazioni di giudizi. Se ad esempio abbiamo come goal corrente:

```
pre   = true
stmt1 =   1 : (pk, sk) = KG ();
stmt2 =   1 : (pk, sk) = KG ();
post  = ={pk,sk}
```

ed in precedenza è stato verificato per i giochi del goal corrente:

```
equiv KG_EQ : GameLH.KG ~ GameRH.KG : true ==> ={res}
```

possiamo usare *call using KG_EQ* per ottenere le verification conditions finali e gestibili dai solver SMT.

4.2.2 Program Transformation Tactics

La seconda tipologia di tattiche abitualmente implementa meccaniche collegate alla manipolazione del flusso di esecuzione del codice e generalmente, se applicate al goal corrente, esse non modificano pre- e post- condition.

Tattica Let - sintassi: $let\ [side][position]ident\ type - expr = prog - expr$
Definisce una nuova variabile fresca nello *stmt* ed alla istruzione specificati. Sovente viene utilizzato nelle formule relazionali di tattiche come *App* per per poter verificare poi Θ.

Tattica IfNeg - sintassi: $ifneg\ [side][position]$
In presenza di un'istruzione condizionale if provvede ad effettuare la negazione dell'istruzione condizionale ed a invertire i due rami, lasciando così la semantica dei programmi invariata.

Tattica Inline - sintassi: $inline\ [side][P_1, ..., P_j|position]$
La tattica provvede ad effettuare l'*inlining* delle procedure $P_1, ..., P_j$; se non sono specificate si considerano le chiamate a procedura nella prima istruzione di ogni *stmt*. Le variabili utilizzate nella procedura sono opportunamente ridenominate per evitare di essere confuse con quelle di LH ed RH.

Tattica Swap - sintassi: $swap\ [side][[num - num]|num]num$
Possiamo effettuare lo spostamento di una serie di istruzioni $[num - num]$

nei programmi: se specifichiamo l'ultimo parametro *num* come positivo, verranno spostate di *num* posizioni verso il bottom, se è negativo, di *num* verso il top. Tale trasformazione è possibile soltanto se le istruzioni scambiate da *swap* sono indipendenti fra loro, ad esempio se abbiamo il goal corrente

```
pre   = true
stmt1 =   1 : x = [0..q-1];
          2 : y = [0..q-1];
          3 : d = B(g^x, g^y, g^(x*y));
stmt2 =
post  = d{1} = (b{2} = b'{2})
```

non possiamo usare la tattica

```
swap{1} [1-1] 2.
```

essendo la terza istruzione dipendente dalla prima.

Tattica Unroll - sintassi: *unroll* [*side*][*position*]
Grazie ad *unroll* possiamo 'estrarre' un'iterazione da un ciclo while, senza che la guardia del ciclo stesso sia verificata, eseguendo una trasformazione nella forma:

$$while\ \ b\ \ do\ \ c \quad \longrightarrow \quad if\ \ b\, then\ \ c; while\ \ b\ \ do\ \ c$$

L'argomento opzionale *position* serve a specificare l'istruzione in cui il ciclo da trasformare si trova.

Tattica Splitw - sintassi: *splitw* [*side*][*position*] : $bool - exp$
Suddivide un ciclo while in due cicli consecutivi, eseguendo una trasformazione in forma:

$$while\ \ b\ \ do\ \ c \quad \longrightarrow \quad while\ \ (b\ \&\&\ bool - exp)\ do\ \ c; while\ \ b\ \ do\ \ c$$

Tale tattica può essere utilizzata per suddividere il ciclo in due parti in modo che successivamente si possa verificare che il primo ciclo (con la tattica *while*) termini con l'invariante opportuna e rimanga una sola iterazione da svolgere per il secondo ciclo while. In questo modo si possono utilizzare le opportune tattiche sul corpo del ciclo e poi effettuare l'unroll sull'ultima iterazione.

Tattiche Condt e Condf - sintassi: $(condt|condf)$ $[side][position]$
Queste tattiche permettono di rimuovere un'istruzione condizionale if o $while$ alla posizione specificata, purché la pre-condition sia tale da verificare la validità del test dell'istruzione condizionale, nel caso di *condt* e della sua negazione nel caso di *condf*.

Usando *condt* vengono generati due subgoal, uno per la verifica del test ed un altro in cui al posto dell'istruzione condizionale vi è il corpo del loop o il primo ramo dell'if; il loop, essendo il test valido, rimane all'istruzione successiva. Nel caso di *condf* invece può essere generato un solo subgoal. Ad esempio, nel caso di un loop in cui la guardia non sia più vera, il ciclo ed il suo corpo possono essere eliminati definitivamente: tale strategia viene impiegata nella pratica per eliminare l'ultima iterazione di un ciclo (vedi tattica *splitw*).

4.2.3 Automatized Tactics

Le tattiche automatizzate utilizzano tecniche euristiche e di analisi statica per combinare le tattiche già descritte e generalmente sono in grado di gestire blocchi di più istruzioni per volta: rappresentano il tentativo di EasyCrypt, ancora in fase di sviluppo, di automatizzare il processo di dimostrazione delle transizioni con pRHL.

Tattica Weakest-Precondition - sintassi: wp $[pos1 - pos2]$
La tattica wp calcola la weakest precondition di frammenti di codice, partendo dal bottom e risalendo fino al top o, se specificato, fino a $pos1$ in LH e $pos2$ in RH. La tattica si ferma quando scopre un'istruzione con un assegnamento probabilistico, un loop o una chiamata a procedura. Consente quindi di gestire in automatico assegnamenti deterministici, generando automaticamente un nuovo subgoal con la post-condition calcolata a partire da quella del goal corrente e tenendo conto della weakest precondition calcolata.

Tattica Strongest-Postcondition - sintassi: sp $[pos1 - pos2]$
Analoga a *wp*, la tattica *sp* consente invece di controllare dal top al bottom i programmi, operando nello stesso modo ed andando a generare un nuovo subgoal con la pre-condition del goal corrente opportunamente modificata.

Tattica Simpl - sintassi: *simpl*

Identica a *wp*, ma tenta anche di risolvere la post-condition risolvendo i casi assurdi o triviali, eventualmente riducendo a *true*.

Tattica Trivial - sintassi: *trivial*
Combina insieme le tattiche *rnd* (one-sided e two-sided) e *wp* per semplificare il goal corrente in modo da gestire frammenti con istruzioni sia deterministiche che probabilistiche. A livello pratico occorre utilizzare *rnd* con opportuni argomenti (fct) per risolvere i casi più complessi di uguaglianza tra espressioni probabilistiche in LH ed RH.

Tattica Auto - sintassi: *auto* $[rel - exp]$
Anche *auto* funziona come *wp* ed è in grado di semplificare game con blocchi di istruzioni deterministiche; nel caso trovi chiamate di procedura in LH ed RH inoltre, prova anche a verificare se esistano giudizi già provati inerenti ad esse per impiegarli, in maniera simile alla tattica di base *call*. Altrimenti può tentare opzionalmente di usare come invariante la $rel - exp$ data come argomento.

4.2.4 Un Esempio: il Cifrario di Vernam

Analizziamo un semplice esempio di dimostrazione riguardante il cifrario di Vernam, anche detto One-Time Pad. La tripla π che lo caratterizza è data dagli algoritmi:

- **Key Generation**: KG genera una chiave simmetrica k in maniera uniformemente distribuita da $\{0,1\}^l$;
- **Enc**: data una chiave k ed un messaggio m dalla stessa distribuzione di probabilità, $Enc(k,m)$ restituisce in output $c = k \oplus m$ con $\oplus$ operatore di XOR fra stringhe di bit;
- **Dec**: data una chiave k ed un crittogramma c dalla stessa distribuzione di probabilità, $Dec(k,c)$ restituisce in output $m = k \oplus c$.

Per dimostrare la nozione di sicurezza perfetta occorre verificare che la distribuzione probabilistica di c, calcolata con Enc suddetto, sia la stessa di c generato probabilisticamente da $\{0,1\}^l$.

Abbiamo perciò i due giochi/esperimenti:

```
game OTP = {
```

```
 var m : message
 var c : ciphertext

 fun KG() : key = { var k : key = {0,1}^l; return k; }

 fun Enc(k:key, m':message) : ciphertext = { return (k ^^ m'); }

 fun Main() : unit = {
  var k : key;
  m = M();
  k = KG();
  c = Enc(k, m);
 }
}.


game Uniform = {
 var m : message
 var c : ciphertext

 fun Main() : unit = {
  m = M();
  c = {0,1}^l;
 }
}.
```

ed il seguente giudizio da dimostrare:

```
equiv Secrecy: OTP.Main ~ Uniform.Main : true ==> ={c,m}.
```

trasformato da EasyCrypt in:

```
pre   = true
stmt1 =   1 : m = M ();
          2 : k = KG ();
          3 : c = Enc (k, m);
stmt2 =   1 : m = M ();
          2 : c = {0,1}^l;
post  = (c{1},m{1}) = (c{2},m{2})
```

Il giudizio viene provato in EasyCrypt come segue:

```
proof.
 inline KG, Enc; wp.
 rnd (c ^^ m{2}); trivial.
save.
```

La tattica inline include il codice di KG ed Enc negli statement e con wp viene computata la weakest precondition, ottenendo il goal seguente:

```
pre   = true
stmt1 =   1 : m = M ();
          2 : k_0 = {0,1}^l;
stmt2 =   1 : m = M ();
          2 : c = {0,1}^l;
post  = (k_0{1} ^^ m{1},m{1}) = (c{2},m{2})
```

A questo punto, avendo due istruzioni probabilistiche possiamo utilizzare la tattica *rnd*: per provare la post-condition del goal corrente forniamo in input una giustificazione appropriata, sotto forma di funzione: $c \oplus m\{2\}$, la quale è una *shortcut* per indicare $c \rightarrow (c \oplus m\{2\})$, ottenendo:

```
pre   = true
stmt1 =   1 : m = M ();
stmt2 =   1 : m = M ();
post  = (forall (x : bitstring{l}),
           x ^^ m{2} ^^ m{2} = x && (x ^^ m{2} ^^ m{2} = x =>
             (x ^^ m{1},m{1}) = (x ^^ m{2},m{2}))) &&
         (forall (y : bitstring{l}),y ^^ m{2} ^^ m{2} = y)
```

La nuova post-condition verifica anzitutto che la funzione data sia un involuzione e poi sostituisce c con x e $c\{2\}$ con $f(x)$ data ossia $x \oplus m\{2\}$.

Infine, rimangono soltanto due assegnamenti random, dato che $M()$ è un operatore probabilistico; la tattica trivial è sufficiente in questo caso e permette di ottenere *stmt* vuoti e la seguente verification condition:

$$true \Longrightarrow \forall x, y \in \{0,1\}^l.(x \oplus y) \oplus y \ = \ x \wedge (x \oplus y, y) \ = \ (x \oplus y, y)$$

risolvibile automaticamente da solver SMT come Alt-Ergo, eventualmente specificato nel codice.

4.3 Prove su Mixnets in EasyCrypt

Nella Sezione 2.3 abbiamo citato diverse mixnets basate su ElGamal come Decryption e Reencryption Mixnets; dato il supporto di EasyCrypt alle prove di CPA-security di ElGamal, si è pensato di utilizzare la sequenza di giochi citata nel paragrafo 3.2.3 come punto di partenza per esprimere

alcune proprietà di base delle mixnets.

4.3.1 Generazione Chiavi per Mixnets ElGamal-Based

Un'aspetto basilare riguardante Decryption ed Encryption Mixnets concerne la generazione delle chiavi per ogni mix server M_i; in un contesto concreto abbiamo infatti una chiave pubblica pk_i ed una chiave segreta sk_i per ogni mix server, nonché una coppia di chiavi (PK, SK) congiunte ottenute come:

$$PK = \prod_{i=1}^{l} pk_i = g^{\sum_{i=1}^{l} x_i}$$

$$SK = \sum_{i=1}^{l} x_i$$

Le chiavi congiunte così ottenute devono poter essere utilizzate per poter crittare e decrittare plaintext correttamente: è stato quindi definito un nuovo gioco rappresentante l'esperimento di CPA indistinguishability, del tutto similare all'originale eccezion fatta per l'algoritmo KG, in modo tale che generasse PK ed SK come chiavi congiunte.

Il codice del gioco elaborato è il seguente, dove il numero di mix server l è dato dalla costante *server_number*:

```
game JOINTKEYGEN = INDCPA where
 KG = {
 (*dichiarazione variabili*)
      ...
     while (i < server_number) {
       x = [0..q-1];
       sk=(sk+x)%%q;
       psk = psk <- (i, x);
       ppk = ppk <- (i, g^x);
       i = i + 1;
     }
     pk=g^sk;
     return (pk, sk);
 }.
```

Come si può notare, un loop è eseguito un numero costante *server_number* di volte e *sk* viene definita come sommatoria modulo q delle chiavi private x_i: questo perchè EasyCrypt supporta ElGamal in versione CPA-sicuro,

ossia con crittogrammi e e chiavi appartenenti al gruppo di Schnorr, un sottogruppo di ordine q di $\mathbb{Z}_p^*$. Le operazioni

```
psk = psk <- (i, x);
ppk = ppk <- (i, g^x);
```

corrispondono alle operazioni di inserimento delle chiavi di ogni mix server in un array: nel programma sono definiti appositamente il tipo array e gli operatori inerenti come inserimento ed estrazione, oltre ad alcuni assiomi per definirne il comportamento, ma è in sviluppo, con le nuove versioni di EasyCrypt, la possibilità di gestire array in maniera automatica come i tipi base.

Per stabilire se il gioco definito fosse negligibly close all'originale esperimento CPA si è utilizzato il seguente giudizio con relativa prova:

```
equiv JD : JOINTKEYGEN.KG ~ INDCPA.KG : true ==> ={res}.
proof.
 wp.
 sp.
 splitw{1}:(i < server_number-1).
 while{1}>>
 : (i{1} <= server_number - 1)
 : server_number - i{1},1.
 trivial.
 condt{1} at 1.
 rnd >>(w -> (w+sk{1})%%q), (w -> (w-sk{1})%%q).
 condf{1} last.
 auto.
save.
```

con il goal iniziale generato da EasyCrypt come segue:

```
pre   = true
stmt1 =   1 : sk = 0;
          2 : i = 0;
          3 : while (i < server_number) {
                x = [0..q - 1];
                sk = (sk + x) %% q;
                psk = psk <~ (i,x);
                ppk = ppk <~ (i,g ^ x);
                i = i + 1;
              }
          4 : pk = g ^ sk;
stmt2 =   1 : x = [0..q - 1];
```

```
post  = (pk{1},sk{1}) = (g ^ x{2},x{2})
```

Le tattiche *wp* e *sp* iniziali consentono di eliminare le istruzioni deterministiche al top ed al bottom di *stmt*1; viene poi usata la tattica *splitw* suddividendo il ciclo while in *stmt*1 in due loop, in modo che al termine del primo valga $i\{1\} = server\_number - 1$: questo viene provato con la tattica while, utilizzando le opportune invariante, variante e bound del ciclo. In questo caso è sufficiente definire variante ed invariante basandosi sulla guardia del loop; a questo punto otteniamo un goal intermedio con la precondition $i\{1\} = server\_number - 1$ voluta:

```
pre   = (exists (x_L, sk_L : int, psk_L : partial_skey, ppk_L :
    partial_pkey,
                 i_L : int),i_L = 0 && sk_L = 0) &&
          i{1} <= server_number - 1 &&
           !(i{1} < server_number - 1 && i{1} < server_number)
stmt1 =    1 : while (i < server_number) {
                 x = [0..q - 1];
                 sk = (sk + x) %% q;
                 psk = psk <~ (i,x);
                 ppk = ppk <~ (i,g ^ x);
                 i = i + 1;
               }
stmt2 =    1 : x = [0..q - 1];
post  = (g ^ sk{1},sk{1}) = (g ^ x{2},x{2})
```

Con la tattica *condt* effettuiamo l'unroll della prima iterazione del loop, su cui possiamo lavorare per giustificare l'equivalenza con la distribuzione probabilistica in *stmt*2. La tattica

```
rnd >>(w -> (w+sk{1})%%q), (w -> (w-sk{1})%%q)
```

viene utilizzata a tale scopo, fornendo la biiezione corretta ed ottenendo il goal:

```
pre   = ((((forall (x : int),
              0 <= x => x <= q - 1 =>
                 (0 <= (x + sk{1}) %% q && (x + sk{1}) %% q <= q
                      - 1) &&
                   ((x + sk{1}) %% q - sk{1}) %% q = x) =>
            (forall (y : int),
              0 <= y => y <= q - 1 =>
                 (0 <= (y - sk{1}) %% q && (y - sk{1}) %% q <= q
                      - 1) &&
```

```
                ((y - sk{1}) %% q + sk{1}) %% q = y) =>
          x{2} = (x{1} + sk{1}) %% q) && 0 <= x{1} && x{1} <= q
              - 1) &&
        0 <= x{2} && x{2} <= q - 1) &&
       (exists (x_L, sk_L : int, psk_L : partial_skey, ppk_L :
           partial_pkey,
                i_L : int),i_L = 0 && sk_L = 0) &&
        i{1} <= server_number - 1 &&
         !(i{1} < server_number - 1 && i{1} < server_number)
stmt1 =  1 : sk = (sk + x) %% q;
         2 : psk = psk <~ (i,x);
         3 : ppk = ppk <~ (i,g ^ x);
         4 : i = i + 1;
         5 : while (i < server_number) {
               x = [0..q - 1];
               sk = (sk + x) %% q;
               psk = psk <~ (i,x);
               ppk = ppk <~ (i,g ^ x);
               i = i + 1;
             }
stmt2 =
post  = (g ^ sk{1},sk{1}) = (g ^ x{2},x{2})
```

Infine, avendo in precedenza effettuato l'unroll dell'unica iterazione rimasta, non rimane altro da fare se non eliminare il loop con *condf* e gestire gli assegnamenti deterministici rimanenti con *auto*: le verification conditions restanti sono automaticamente risolvibili dai solver Alt-Ergo o CVC3.

Abbiamo provato il giudizio sulle funzioni KG e possiamo provare il giudizio complessivo sui game INDCPA e JOINTKEYGEN:

```
equiv JD2 : JOINTKEYGEN.Main ~ INDCPA.Main : true ==> ={res}.
proof.
 app 1 1 ={pk,sk}.
 call using JD.
 trivial.
 auto.
 trivial.
 auto.
save.
```

Sono impiegate le strategie *app* e *call* usando il giudizio JD dimostrato in precedenza per stabilire l'identica distribuzione probabilistica di (pk, sk) nei due giochi.

Un ulteriore game è stato aggiunto per rappresentare l'esperimento in

cui un avversario concreto, definito come la funzione Mix_Server_Adv e simile al classico avversario CPA, ma che possiede un'ulteriore informazione del cifrario, ovvero una chiave privata x_i. Tale ruolo di avversario può essere ricoperto, nel contesto pratico delle mixnets, da un qualunque mix server M_i. Viene definito perciò il gioco:

```
game MIX_SERVER_EXP ={

  (*def. di Enc come in JOINTDEC *)
      ...

  fun KG2():pkey*skey*partial_skey = {
    (*corpo identico a KG in JOINTDEC *)
      ...

    return (pk, sk, psk);
  }
  fun Main() : bool = {
    (*dichiarazione variabili*)
     ...
    (pk,sk,psk) = KG2();
    (m0,m1,sigma) = A1(pk);
    b = {0,1};
    c = Enc(pk, b ? m0 : m1);
    k=[0..server_number];
    mixnet_priv_key=psk->k;
    b' = Mix_Server_Adv(c, sigma,mixnet_priv_key);
    return (b = b');
  }
}.
```

La funzione $KG2$ è in realtà identica a KG ma restituisce anche l'array delle chiavi private parziali, ognuna delle quali appartiene ad un solo mix server. Volendo dimostrare che l'avversario/mix server M_i ha in realtà le stesse possibilità di un avversario CPA dato che ha a disposizione solo una chiave privata parziale dalla quale non è possibile dedurre nulla sulla chiave sk, proviamo con EasyCrypt il giudizio:

```
equiv JD3 : MIX_SERVER_EXP.Main ~ JOINTKEYGEN.Main : true ==> ={
    res}.
proof.
 inline Mix_Server_Adv.
 auto.
```

```
 trivial.
 inline.
 auto.
 sp.
 while>> :(={i} && i{1} <= server_number - 1).
 trivial.
 trivial.
 auto.
save.
```

Descriviamo brevemente lo svolgimento della prova: si effettua l'inlining della funzione Mix_Server_Adv, la quale si riduce a invocare l'avversario $A2$ definito in precedenza, e si impiegano *auto* e *trivial* per calcolare le weakest precondition e gestire gli assegnamenti random presenti. Un ulteriore inlining viene effettuato per le funzioni $KG2$ e KG, le quali essendo identiche, eccetto che per i valori di ritorno, possono essere verificate semplicemente con la tattica *while* e con *trivial* per le istruzioni probabilistiche. Le restanti istruzioni deterministiche a questo punto sono identiche e possiamo terminare la prova con *auto*.

Concettualmente, abbiamo aggiunto due giochi al di sopra del game INDCPA rappresentato in Fig. 3.2 e sfruttato le proprietà di ElGamal supportate da EasyCrypt, i cui formalismi sono codificati in CoQ. Ulteriori considerazioni saranno espresse nel Capitolo 5.

4.3.2 Prove su Omomorfismo di ElGamal

Nel paragrafo 1.2.4 abbiamo citato le proprietà di omomorfismo moltiplicativo di ElGamal che sono alla base di molti protocolli: dalle proprietà di reencryption alle prove ZK sul prodotto aggregato degli input/output delle mixnets.

Questa proprietà è esprimibile sotto forma di sequenze di giochi: esplicitiamo un esperimento in cui vengono generati e crittati con ElGamal due input m_0, m_1 e ne viene restituita la moltiplicazione omomorfica dei crittogrammi, mentre in un secondo esperimento m_0, m_1 vengono moltiplicati fra loro e poi crittati, restituendo il ciphertext risultante. In EasyCrypt abbiamo così il primo game

```
game VERIFIER_GAME = {

  fun KG() : pkey * skey = {
```

```
    var x : int = [0..q-1];
    return (g^x, x);
  }
 fun MG() : plaintext * plaintext = {
    var x : int = [0..q-1];
    var y : int = [0..q-1];
    return (g^x, g^y);
  }
  fun PG() : int * int = {
    var x : int = [0..q-1];
    var y : int = [0..q-1];
    return (x, y);
  }
  fun Enc(pk:pkey, m:plaintext, y: int): group = {
    return (m* (pk^y));
  }

  abs A1 = A1 {}

  fun Main() : group = {
  (dichiarazione variabili)
        ...

    (pk,sk) = KG();
    (m0,m1) = MG();

    (r0,r1)=PG();

    b0=Enc(pk,m0,r0);
    b1=Enc(pk,m1,r1);

    return ( (b0*b1) );
  }
}.
```

dove *PG* esplicita la generazione dei padding random effettuata in altre dimostrazioni direttamente in *Enc*; nell'ambito di prove ZK parliamo di un esperimento in cui il valore di ritorno può essere calcolato semplicemente utilizzando i crittogrammi, senza ulteriori conoscenze e perciò eseguibile da un qualunque *verifier* esterno. Il secondo game è definito come:

```
game PROVER_GAME = VERIFIER_GAME where
  Main = {
    (dichiarazione variabili)
```

```
        ...

    (pk,sk) = KG();
    (m0,m1) = MG();

    (r0,r1)=PG();

    ch=Enc(pk,m0*m1,r0+r1);
    return (ch);
  }.
```

Questo game, simile al precedente, in questo caso è riferito ad un esperimento eseguibile dal *prover*, nel nostro caso ogni server M_1, il quale può calcolare il prodotto aggregato utilizzando direttamente i padding random r_0, r_1 da esso stesso generati. La prova che i due giochi sono negligibly close è la seguente:

```
equiv Pr0 : VERIFIER_GAME.Main ~ PROVER_GAME.Main : true ==> ={
    res}.
proof.
 app 1 1 ={pk,sk}.
 call.
 trivial.
 app 1 1 ={m0,m1,pk,sk}.
 call.
 trivial.
 app 1 1 ={r0,r1,m0,m1,pk,sk}.
 call.
 trivial.
 inline.
 auto.
save.
```

Le tattiche *app*, *call* e *trivial* sono usate di volta in volta per le procedure KG, MG, PG stabilendo l'identica distribuzione di probabilità delle variabili random generate nei giochi ed effettuando infine l'inlining della procedura *Enc*. Per risolvere l'ultimo goal con *auto* è necessario definire nel programma il seguente assioma sull'omomorfismo moltiplicativo, verificato automaticamente da EasyCrypt:

```
axiom Homomorphic_enc:
forall (m0,m1:plaintext,y:pkey, r0,r1:int), (m0*y^r0)*(m1*y^r1)
    = (m0*m1)*y^(r0+r1).
```

Questa semplice prova può essere, come verrà descritto nel prossimo, il primo passo per una prova di sicurezza di una specifica proposta di mixnet, per verificare le permutazioni effettuate da ogni mix server.

Capitolo 5

Conclusioni

5.1 EasyCrypt e Mixnets: Criticità

Il lavoro svolto è stato abbastanza peculiare ed inedito in quanto si proponeva di analizzare un tool software come EasyCrypt per verificarne l'adattabilità ad un contesto pratico come quello delle mixnets. Sebbene le diverse proposte di mixing networks in letteratura fossero generalmente definite in termini teorici piuttosto che implementativi, si è potuto constatare come in realtà si trattasse di tecniche elaborate per essere effettivamente sviluppate, con tanto di indicazioni protocollari, in contesti pratici come quelli citati nel Capitolo 2, ad esempio quello dell'e-voting.

EasyCrypt, ora alla versione 0.2 ed ancora in stadio di sviluppo, si è dimostrato sufficientemente versatile per esprimere alcuni semplici aspetti legati alle mixnets; il linguaggio pWhile è infatti un linguaggio imperativo di potere espressivo comparabile a quello di molti linguaggi *general purpose*, con l'aggiunta di costrutti specifici ed indispensabili per le dimostrazioni crittograficche. Inoltre pWhile è anche estensibile, dando la possibilità di definire tipi, operatori ed assiomi come visto nel Capitolo 4. La difficoltà maggiore riguarda il fatto che, per implementare nuove funzionalità complesse, occorre che il tool venga modificato e riadattato dagli sviluppatori per formalizzare la teoria matematica alla base dei nuovi costrutti, come è stato fatto per la teoria dei gruppi di ElGamal.

A livello teorico rimane anche la necessità di definire tecniche e protocolli legati alle mixnets in termini di sequenze di giochi, possibilmente in maniera rigorosa e non in maniera intuitiva come nel Capitolo 4: anche questo *task* in letteratura è trascurato e riservato invece a costruzioni elementari come

i cifrari, di cui del resto sono disponibili esperimenti e nozioni di sicurezza già formalizzate, e risultando quindi più facilmente 'traducibili' in giochi.

EasyCrypt in questo senso non è al momento in grado di fornire supporto; come specificato dagli sviluppatori in [27], il tool non implementa alcun meccanismo per aiutare gli utenti a costruire i game intermedi, partendo dai giochi di cui si vogliono studiare le proprietà di sicurezza.

Infine occorre notare che alcuni limiti, tipici di un software ancora in via di sviluppo come EasyCrypt, sono dovuti a *bugs*, scarsa ottimizzazione ed automatizzazione e problemi di supporto alle librerie ma possono essere risolti con il rilascio di nuove versioni, approfittando anche dell'aiuto di una comunità di utenti poco numerosa ma attenta e competente come quella di coloro che studiano e propongono prove formali crittografiche, in ambito accademico e non.

5.2 Sviluppi Futuri

Uno degli obiettivi più immediati che è stato individuato è stato quello di usare EasyCrypt per una prova formale delle proprietà di sicurezza delle mixnets relativamente alla permutazione degli elementi. Una proposta interessante in questo senso è la già citata mixnet [16] di Jakobsson e Juels. Ogni *permutation network* in grado di eseguire una permutazione di n elementi in input ad una mixnet, può essere infatti costruita usando ricorsivamente dei sorter *2-by-2* [6], perciò è sufficiente implementare un sorter 2-by-2 efficiente e sicuro per ottenere una permutation network completa.

In Millimix viene descritto un protocollo in cui ogni mix server M_i, mediante un sorter 2-by-2, deve provare la corretta permutazione di una coppia di input (α_1, β_1) e (α_2, β_2), crittogrammi dei plaintext m_1, m_2; gli output saranno (α'_1, β'_1) e (α'_2, β'_2), crittogrammi di (m'_1, m'_2), e perciò M_i dovrà dimostrare che vale $(m_1, m_2) = (m'_1, m'_2)$ oppure $(m_1, m_2) = (m'_2, m'_1)$. Per dimostrare tale proprietà occorre provare due uguaglianze:

- **Eq1**: $m_1 = m'_1$ oppure $m_1 = m'_2$;
- **Eq2**: $m_1 m_2 = m'_1 m'_2$

Per dimostrare Eq1 ed Eq2 sono proposti rispettivamente due sottoprotocolli $\mathcal{DISPEP}$ e $\mathcal{PEP}$ basati sul protocollo di identificazione di Schnorr. Il protocollo $\mathcal{PEP}$ è in grado di stabilire se un crittogramma (α', β') sia

reencryption di un crittogramma (α, β): per provare Eq2 un mix server può calcolare così $Enc(m_1 m_2)$ ed $Enc(m'_1, m'_2)$ e dimostrare che il secondo è la reencryption del primo: un verifier esterno può anch'egli calcolare $Enc(m_1 m_2)$ ed $Enc(m'_1, m'_2)$ utilizzando le proprietà omomorfiche di ElGamal e dimostrate come descritto nel paragrafo 4.3.2; il lavoro svolto diventa perciò una base di partenza per una dimostrazione più estesa comprensiva di giochi che descrivono il funzionamento del protocollo di Millimix.

Più in generale, è necessario seguire l'evoluzione del tool di EasyCrypt il quale allo stato attuale, Marzo 2013, sta per subire una consistente reimplementazione: nelle versioni *preview* disponibili pubblicamente si può notare la presenza di moduli esterni denominati *theories* che ridefiniscono alcuni tipi di dato come *int*, *boolean* e *list* e ne definiscono di nuovi come ad esempio il tipo *array*. Nuove funzionalità offerte dovranno quindi essere testate e dovrà essere verificata la retrocompatibilità con EasyCrypt 0.2 affinché le prove illustrate siano riutilizzabili con le nuove versioni.

5.3 Ringraziamenti

Un caloroso ringraziamento va anzitutto a Jessica, il cui supporto morale è stato determinante per il lavoro di tesi, senza considerare le correzioni di bozze, i suggerimenti linguistici e di stile. Soprattutto, grazie di cuore per l'aiuto che mi hai dato durante tutto il percorso di laurea, Jessica: sono certo che senza il tuo conforto nei momenti più difficili ed il tuo incoraggiamento non avrei potuto ottenere i risultati che ho conseguito.

Un grazie va anche alla mia famiglia che mi ha sostenuto e dato la possibilità, anche economicamente, di seguire un corso di studi che è stato un'importante esperienza di formazione, non solamente culturale.

Vorrei ringraziare poi tutti coloro che mi hanno aiutato nel lavoro di tesi per la disponibilità e la competenza da loro offertami, a partire dal mio relatore e docente di Crittografia Ugo Dal Lago, che ha suscitato il mio interesse per la materia e mi ha più volte fornito le indicazioni giuste per comprendere al meglio come impostare il percorso di ricerca.

Ringrazio infine il team di sviluppatori di EasyCrypt: Gilles Barthe, François Dupressoir, Benjamin Grégoire, Pierre-Yves Strub ed in particolare Santiago Zanella Béguelin, il cui aiuto mi è stato indispensabile per portare a termine le prove e comprendere il funzionamento di EasyCrypt.

Bibliografia

[1] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[2] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO*, pages 307–315, 1989.

[3] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[4] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[5] Krishna Sampigethaya. A Survey on Mix Networks and Their Secure Applications. volume 94. Proc. of IEEE Computer Society, December 2006.

[6] Ben Adida. *Advances in cryptographic voting systems*. PhD thesis, Cambridge, MA, USA, 2006.

[7] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct rsa-implementation of mixes. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, pages 373–381, 1989.

[8] K. Itoh C. Park and K. Kurosawa. Efficient anonymous channel and all/nothing election scheme. In *Advances in Cryptology-Eurocrypt*, pages 248–259. Springer-Verlag, 1994.

[9] Philippe Golle, Markus Jakobsson, Ari Juels, and Paul F. Syverson. Universal re-encryption for mixnets. In *CT-RSA*, pages 163–178, 2004.

[10] Markus Jakobsson and Ari Juels. An optimally robust hybrid mix network. In *PODC*, pages 284–292, 2001.

[11] Markus Jakobsson. A practical mix. In *EUROCRYPT*, pages 448–461, 1998.

[12] Markus Jakobsson. Flash mixing. In *PODC*, pages 83–89, 1999.

[13] Yvo Desmedt and Kaoru Kurosawa. How to break a practical mix and design a new one. In *EUROCRYPT*, pages 557–572, 2000.

[14] Masayuki Abe. Universally verifiable mix-net with verification work indendent of the number of mix-servers. In *EUROCRYPT*, pages 437–447, 1998.

[15] Masayuki Abe. Mix-networks on permutation networks. In *ASIACRYPT*, pages 258–273, 1999.

[16] Markus Jakobsson and Ari Juels. Millimix: Mixing in small batches. Technical Report 99-33, Certer for Discrete Mathematics & Theoretical Computer Science(DIMACS), 1999.

[17] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, pages 339–353, 2002.

[18] Philippe Golle, Sheng Zhong, Dan Boneh, Markus Jakobsson, and Ari Juels. Optimistic mixing for exit-polls. In *ASIACRYPT*, pages 451–465, 2002.

[19] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, Oktober 1949.

[20] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.

[21] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, pages 409–426, 2006.

[22] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. `http://eprint.iacr.org/`.

[23] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[24] Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2010.

[25] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[26] Santiago Zanella Béguelin Juan Manuel Crespo Cesar Kunz Gilles Barthe, Benjamin Grégoire. *The EasyCrypt Tool: Documentation And User's Manual*, September 2012. `http://software.imdea.org/projects/certicrypt/easycrypt-0.2.pdf`.

[27] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In *3rd International Conference on Interactive Theorem Proving, ITP 2012*, pages 12–27. Springer, 2012.